Assembly Language Programming

Objective Of This Module

This module offers an introduction to assembly language programming and the Atari Assembler Editor. Through the activities in this module you will see how assembly language is a particularly good language for fast, smooth animation. You also will find that assembly language requires programming in great detail. Upon completing this module, you will not be prepared to write an arcade game. Just as the novice pianist cannot hope to be able to write a piano concerto after two weeks of practice, a novice assembly language programmer cannot possibly program a PAC MAN game in two weeks. In fact, professional programmers with years of experience take six to eight months to produce an arcade game. Hopefully, you will find the rewards of a successful program are well worth the hard work.

Overview

1. The Assembler.
   What is the assembler and what does it do?

2. Assembly Language Format.
   What is the correct syntax and punctuation for assembly language programs?

3. Instructions and Beginning Addressing Modes.
   This section offers you an opportunity to experiment with various assembly language instructions.

4. Indexed Addressing Modes.
   The eight different addressing modes available on the Atari are explained and demonstrated.

5. Animation.
   In this section you will write assembly language subroutines that move a spinning pinwheel around on the screen with a joystick.

Prerequisite Concepts

1. You must have completed the Machine Architecture Module before doing this module.

Materials Needed

1. An Atari Assembler Editor Cartrigde and the User Manual.
2. An Advanced Topics Diskette.

The Assembler

This section explains how assembly language programs are executed and the assembler editor's role in the process.

In the Machine Architecture Module you recently completed, you had a chance to see some assembly language instructions and learn how the 6502 executes a program. You also learned that, regardless of what language you are programming in, the 6502 only understands machine code. How then does assembly language get converted to machine code in order for the CPU to execute your program?

Writing and executing assembly language programs requires an "assembler editor." You have already used the Atari Assembler Editor cartridge to execute the assembly language programs in the Machine Architecture Module. When you insert your assembler cartridge in the Atari and turn on the computer, two programs on a chip inside the cartridge are loaded into memory. One of the programs, called the "assembler," is responsible for converting your assembly language program to machine code. The second program, called the "editor," enables the programmer to type and edit the assembly language program before it is "assembled" to machine code by the assembler.

The assembly language program that a programmer writes and types into the computer is called the "source code." The programmer uses the editor to insert, delete, or alter any part of the source code. The source code includes the three letter assembly language instructions, variable names, memory addresses, and labels. Listed below is the source code for a program that prints an arrow in the upper left hand corner of the screen. The program simply loads the accumulator with the code number for an arrow, $7D. The $7D is then stored in screen RAM in order to print the arrow on the screen.

```
*=$0600      ;ORIGIN OF PROGRAM
LDA #$7D     ;LOAD ACCUMLATOR WITH CODE FOR AN ARROW
STA $9C40    ;SCREEN RAM LOCATION
RTS          ;RETURN FROM SUBROUTINE
```

If you look at the right hand side of the program, you will notice that the source code includes remarks and explanations about what the program does. These comments are comparable to REM statements in BASIC. In assembly language you use a ";" to indicate that a remark follows, the same way you use a REM in BASIC. However, comments in assembly language are much more vital than in BASIC because of the difficulty people have understanding assembly code.

Before this assembly language program can be executed, it must be passed through the assembler. The assembler reads through the source code and converts the program to a numerical code which the microprocesssor can understand. (The assembler ignores the comments because they are not pertinent information to the CPU. The comments are only useful to the human reader who is trying to understand the program.) The result is called the "object code." If you look to the left of the source code in the diagram below, you will see the object code. Note that the object code is listed in hexadecimal. The object code is also called the "machine code."

```
Object Code             Source Code

0000             0100   *=$0600        ;ORIGIN OF PROGRAM
0600 A97D        0110   LDA #$7D        ;LOAD ACC. WITH ARROW
0602 8D409C      0120   STA $9C40      ;SCREEN RAM LOCATION
0605             0130   RTS            ;RETURN FROM SUBROUTINE
```

As the assembler converts the source code to object code, it stores the hexadecimal values in successive memory locations. The first instruction of the program, *=$0600, instructs the assembler to store the object code in memory starting at $600. The column on the far left of the object code above holds the addresses of where the object code is stored in memory. The numbers just to the right of the memory addresses comprise the object code, which has been stored in memory. For a closer look at how the object code has been stored in memory, see the diagram below.

```
Object Code in    Source Code
   Memory

$600  | A9 | <--- LDA #$7D     ;LOAD THE ACC. WITH ARROW
$601  | 7D | <---------
$602  | 8D | <---STA $9C40     ;STORE ACC. IN SCREEN RAM
$603  | 40 | <-----------
$604  | 9C | <-----
$605  | 60 | <---RTS           ;RETURN
```

A code number called the "opcode" has been stored in memory for each instruction. For example, A9 is the opcode for the LDA instruction. The CPU recognizes the A9 as a "load the accumulator" instruction. The opcodes are called opcodes because they are the "code" numbers that tell the microprocessor which "operation" to perform. The 8D (STA) in memory location $602 instructs the CPU to store the value in the accumulator into the specified location. All opcodes are one byte in length, so they take up one memory location.

The number following an instruction in the source code is called the "operand." It is called the operand because it is the number the CPU will be "operating" on when it executes the instruction. For example, the $7D following the LDA is the number the CPU will load into the accumulator. This will be explained in more depth in the next section. However, note that the operand is stored in memory directly after the opcode for the instruction. Also note that the entire object code is listed in hexadecimal numbers.

To summarize, the assembler converts the source code, or English-like version of the program, to object code. The object code is the hexadecimal version of the program, which the assembler stores in memory. It is also referred to as the machine code. The object code is the specific set of instructions that the microprocessor will execute. Turn to Assembly Language Programming Worksheet #1 to have a closer look at some source code and object code.

Assembly Language Programming Worksheet #1

You will need an Assembler Editor Cartridge and an Advanced Topics Diskette to complete this worksheet.

1. Boot up your system with the advanced topics diskette and the Assembler Editor Cartridge. You should have the "EDIT" prompt in the upper left hand corner of your screen. First, ENTER the "ARROW" program from your advanced topics diskette into your computer.

   Type: ENTER #D:ARROW

2. Now type LIST. What type of code do you see? _________

3. To execute the program, the source code must be converted to object code by the assembler.

   Type: ASM and press RETURN

   The combined source code and object code should scroll up on the screen. The code you see on the screen should be the same as the code listed below.

```
0000        0100     *=    $600     ;ORIGIN OF PROGRAM
0600 A97D   0110     LDA #$7D       ;LOAD ACC. WITH ARROW
0602 8D409C 0120     STA $9C40      ;SCREEN RAM LOCATION
0605 60     0130     RTS            ;RETURN FROM SUBROUTINE
```

4. We know that the opcode for LDA is A9 and the opcode for STA is 8D. What is the opcode for RTS?_____

5. Now let's run the program.

   Type: BUG and press RETURN

   You should see the word "BUG" on the screen. The Atari Assembler Editor executes the program from the "debugger." The debugger is another program on the assembler cartridge; it enables you to look at or change the contents of specific memory locations. Don't worry if you don't understand this. However, if you would like to learn more about how to use the debugger, see chapter 5, "Using the Debugger," of the Assembler Editor User's Manual.

6. Now you must press the SHIFT and CLEAR keys at the same time. This clears the screen. If you executed the program with an instruction at the bottom of the screen, once the program had been executed, the screen would scroll up and arrow will no longer be visible.

Type: SHIFT/CLEAR

7. To execute the program you have to tell the computer where the object code is stored in memory.

Type: G600   and press RETURN

The program is stored at memory location 600. So we use the "G" or GO command to tell the computer to execute the program that begins at $600

8. Try changing the character printed on the screen to another character by completing the steps below. First, you must return to the editor.

Type: X and press RETURN

To see the source code again,

Type: LIST and press RETURN

By holding down the "CTRL" key while pressing one of the arrow keys, you can move your cursor up to edit your source code. Place the cursor over the 7 in the #$7D, following the LDA instruction. Type in another number and press RETURN. Then go back to the debugger,to execute the program, by typing BUG. Type SHIFT/CLEAR, to clear the screen before typing "G600" to execute the program. The values for the internal character set are listed at the back of this module if you want to experiment with putting specific letters on the screen. The values are listed in decimal, so you must convert them to hexadecimal to use them in this program.

9. To see how fast the CPU is putting the arrow on the screen, you can run a program called ARW2 on the Auxiliary Advanced Topics Diskette. See your instructor for a copy of the disk. ENTER the ARW2 program.

Type: ENTER ARW2

The ARW2 program loads the accumulator with the value for an arrow, and then stores it in screen RAM, just as the ARROW program did. However, the ARW2 program stores a zero in screen RAM where the arrow was placed to show how fast the arrow is displayed and then erased. Assemble the program and

Type: ASM   and press RETURN

Type: BUG   RETURN and G600

Did you see it? ____ Probably not. This short assembly language program is executed so quickly, you can't even see the arrow displayed.

Once the source code has been assembled to object code and the object code is stored in memory, how does the computer go about executing the program? You may remember from the Machine Architecture module that the CPU can only execute one instruction at a time. To compensate for this the program is stored in memory and the CPU "fetches" one instruction at a time from memory. The CPU goes through a repeated cycle of fetching instructions one at a time and executing them until the entire program has been completed. The actual set of steps the microprocessor takes to execute a program is called the "fetch cycle."

Fetch Cycle

1. Fetch an instruction from memory. Get the opcode and an accompanying operand if there is one.

2. Advance the program counter to the address of the next instruction to be executed.

3. Execute the instruction.

4. Return to #1 and start the cycle over again.

First, the CPU fetches the instruction to be executed. Before executing the instruction, however, the CPU advances the program counter, a two byte register in the CPU, to the address of the next instruction to be executed. Then the CPU executes the instruction it had previously fetched. When the first instruction is completed, the CPU starts the cycle over again. The program counter holds the address of the next instruction to be executed. A new instruction is fetched and the program counter is advanced again. Read along as we execute the fetch cycle with the ARROW program.

1. Fetch the instruction. The CPU fetches the first instruction of the program from memory. It knows where the first instruction is, because you gave it the starting address of he program when you typed "G600". When the CPU fetches the instruction from memory, it gets both the opcode and the operand. In the ARROW program the CPU fetches both A9 and 7D. The opcode A9 is the signal to the CPU to also fetch the value in the next memory location. Opcodes not only instruct the CPU on what type of operation to perform, they also indicate to the CPU how many bytes in memory are associated with that instruction. This will become clearer as you proceed through the module. Look at Diagram 1 below. The CPU is holding the A97D (LDA #$7D) command.

2. *Advance the program counter.* Before the A97D (LDA #$7D) is executed, the program counter must be advanced to the address of the next instruction to be executed. The next instruction of the ARROW program is the 8D (STA), which is in memory location $602. Put the address of the 8D instruction in the program counter in Diagram 1.

3. *Execute the instruction held in the CPU.* Now execute the load command (A97D). Load the accumulator in Diagram 1 with $7D.

4. *Return to #1 and repeat the cycle.* Continue with the explanation of the fetch cycle below.

Diagram 1

```
Source Code   Object Code        6502 Processor

  *=$0600
  LDA #$7D        $600 | A9 |       COMMAND  | A97D |
                  $601 | 7D |
  STA $9C40       $602 | 8D |       PROGRAM COUNTER  |      |
                  $603 | 40 |
                  $604 | 9C |
  RTS             $605 | 60 |       ACCUMULATOR  |      |
```

1. *Fetch the next instruction.* The CPU fetches the next instruction based on the address in the program counter. The program counter has $602, so the CPU fetches the 8D (STA) instruction. This time the CPU fetches the two bytes in memory following the 8D in order to get the entire "store" command (STA $9C40). The 8D was a signal to the CPU that the instruction was a store instruction and that the operand was two bytes. The reason the operand is two bytes in this case is that the operand is the address of screen RAM ($9C40) and all addresses are two bytes. You may have noticed that the two bytes of the address have been reversed, so that the low order byte,40, is stored in memory before the high order byte, 9C. At this point it is not necessary for you to understand why the CPU does this. Just remember that whenever an address is stored in memory, the two bytes of the address are reversed. If you look at Diagram 2 below, you will see that the CPU holds the entire store command (8D409C).

2. *Advance the program counter.* The next instruction in the ARROW program is the RTS (60). Place the *address* of the opcode 60 in the program counter in Diagram 2 before executing the previously fetched instruction.

3. Execute the instruction. Now the "store" command in the CPU is executed. In the Diagram below execute the instruction by storing the value in the accumulator in $9C40. When the arrow has been stored in screen RAM, it appears on the screen.

4. Return to #1 and repeat the cycle. Continue with the last fetch cycle of executing the ARROW program below.

Diagram 2

```
Source Code    Object Code       6502 Processor

  *=$0600
  LDA #$7D       $600 | A9 |        COMMAND   | 8D409C |
                 $601 | 7D |
  STA $9C40      $602 | 8D |        PROGRAM COUNTER   |        |
                 $603 | 40 |
                 $604 | 9C |
  RTS            $605 | 60 |        ACCUMULATOR | 7D |
```

1. Fetch the next instruction. The address in the program counter is $605, so the opcode for RTS in $605 needs to be fetched. RTS is an instruction that does not require an operand. Consequently, the CPU only fetches one byte. The command the CPU fetches will always be one, two, or three bytes long. The CPU knows how many bytes to fetch from memory based on the opcode of the instruction. Place the opcode for the RTS instruction in the command holder in the 6502 in Diagram 3 below.

2. Advance the program counter. Since the ARROW program does not contain any more instructions after the RTS instruction, the program counter is reset to the address of the next instruction in the assembler to be executed. If the ARROW program had been initiated from another program, the program counter would return to the address of the last instruction executed in the original program. For example, the MESSAGE program in the Machine Architecture Module ran an assembly language program from a BASIC program. When the assembly language routine was completed, the BASIC program continued.

3. Execute the instruction. Since we ran the ARROW program from the debugger, the CPU returns to the debugger. The ARROW program has been successfully completed.

Diagram 3

```
     Source Code  Object Code        6502 Processor

       *=$0600
       LDA #$7D      $600 | A9 |       COMMAND □
                     $601 | 7D |
       STA $9C40     $602 | 8D |       PROGRAM COUNTER □
                     $603 | 40 |
                     $604 | 9C |
       RTS           $605 | 60 |       ACCUMULATOR □
```

The computer is truly an amazing machine, but let's see if we can trick it by putting the value of an opcode into the position of an operand. Turn to Assembly Language Programming Worksheet #2.

Assembly Language Programming Worksheet #2

You will need an Assembler Editor cartridge and an advanced topics diskette to complete this worksheet.

1. Boot up the system and enter the ARROW program.

   Type: ENTER #D:ARROW

2. LIST the program and assemble it.

   Type: ASM and press RETURN

3. Note that the object code is listed by commands. So the two bytes for the LDA #$7D command are listed on one line (600 A97D). The next line contains the three bytes for the entire STA $9C40 command (602 8D409C). And the one lone byte for the RTS command appears on the last line of the object code (605 60). When the A9 is in the position of the opcode, which is the first byte of the command, the computer knows that the A9 represents a load instruction. The computer also knows that the opcode is followed by a one byte operand. However, what would happen if you put an A9 in the position of an operand (LDA #$A9)?

4. LIST the program again. Using the CTRL key in conjunction with the arrow keys, place the cursor over the 7 in the LDA #$7D command. Replace the 7D with A9.

   Type: A9 and press RETURN

   Press BREAK a few times to get below the listing of the program before assembling the program.

5. Assemble the program.

   Type: ASM and press RETURN

   The first line of the object code should read: 600 A9A9. The first A9 is the opcode for the LDA instruction. What will the computer do with the A9 in the operand? Run the program.

   Type: BUG and press RETURN

   Type: SHIFT/CLEAR

   Type: G600

When you run the program, you should see an inverse "I". A9 is the internal character set code for that letter.

When a value is in the position of an instruction in the object code, the CPU treats the value as an instruction. Conversely, when the value is in the position of an operand in the object code the computer treats the value as an operand. In this program the operand is used as a letter to be printed on the screen. Thus, the opcode A9 tells the computer to load the accumulator with the value in the operand, which also happens to be an A9, and represents an inverse "I".

```
Instruction
   Opcode     Operand
         \    /
     0600 A9A9   0110      LDA #$A9  ;LOAD ACCUMULATOR
     0602 8D409C 0120      STA $9C40  ;STORE A9 ON SCREEN
```

Assembly Language Format

You have no doubt noticed that the source code of assembly language programs has a unique and structured format. The source code contains information in columns or "fields." There are three fields: the label field, the command field, and the comment field. Each field is separated from the next with a space. The label field and the comment field are optional.

Source Code Fields

Label	Command	Comment
BEGIN	LDA #$7D	;LOAD ACC. WITH AN ARROW

The Label Field

A label enables the programmer to assign a name to a command or to the beginning of a subroutine. A label must begin with a letter (A-Z), and it can only contain letters, numbers, and periods. It is good practice to make labels descriptive, but also try to limit them to no more than eight characters.

Suppose we put the label BEGIN in front of the LDA #$7D command in the ARROW program. And instead of having an RTS instruction at the end of the program, we replace it with a "JMP" instruction. A JMP instruction enables you to "jump" to a label. Look over the listing below. What do you think the program will do? ____

____

```
        *=$0600
BEGIN   LDA #$7D     ;LOAD ACC. WITH AN ARROW
        STA $9C40    ;SCREEN RAM
        JMP BEGIN    ;DO IT AGAIN
```

Turn to Assembly Language Programming Worksheet #3 to see how to insert a label into the ARROW program and see what this new program does.

Assembly Language Programming Worksheet #3

1. ENTER the ARROW program on the advanced topics diskette.

```
     Type: ENTER #D:ARROW
```

2. LIST the program. Use the CTRL and arrow editing keys to place the cursor directly over the <u>space</u> <u>before</u> the LDA instruction.

3. While holding down the CTRL key, press the insert key (in the upper right hand corner of the keyboard) five times - once for each letter in the word BEGIN. Be sure there is a space between the label and the command.

```
    Type:  BEGIN   and press RETURN
```

4. Using the CTRL and arrow editing keys again, move the cursor down over the "R" in the RTS instruction.

```
     Type:  JMP BEGIN  and press RETURN
```

Your listing should look like this.

```
        0100  *=$0600
        0110 BEGIN LDA #$7D     ;LOAD ACC. WITH AN ARROW
        0120  STA $9C40    ;SCREEN RAM
        0130  JMP BEGIN    ;DO IT AGAIN
```

The numbers on the left are the decimal line numbers. They are there strictly for editing purposes. Just as in BASIC, every line of code must have a line number, and you can delete or insert lines using line numbers.

5. Assemble and run the program.

```
     Type:  ASM   and press RETURN
     Type:  BUG   and press RETURN
     Type:  G600
```

6. You have created an infinite loop. You didn't have to type SHIFT/CLEAR because the infinite loop prevents the screen from scrolling. To stop the program you must press the BREAK key.

The label field is always separated from the command field with a space. If no label is being used, you must leave a space between the line number and the command field. The space indicates to the assembler that no label is being used.

The Command Field

The "command" field follows the label field. The command field includes the instruction and the operand. The three letter instructions are also referred to as "mnemonics."

```
                    Command Field

                   mnemonic operand
                        LDA #$7D
```

There is always one space between the mnemonic and the operand in the command field.

The Comment Field

The third field is the "comment" field. Comments are optional but highly recommended. You will find in assembly language programming that even though you may know a program inside and out when you write it, when you go back to it a few days later, you will struggle to remember exactly how the program works if the code is not well documented.

Comments are separated from the other fields with a ";". Comments can follow the command field or you can start a line with a ";" and devote the entire line to a comment.

```
0100 ;THIS PROGRAM PRINTS WHATEVER CHARACTER
0110 ;IS STORED IN THE ACCUMULATOR ONTO THE
0120 ;GRAPHICS 0 SCREEN.  THE VALUES FOR
0130 ;THE INTERNAL CHARACTER SET ARE USED
0140 ;TO STORE A CHARACTER IN SCREEN RAM.
0150 ;
0160  *=$0600
0170 BEGIN LDA #$7D   ;LOAD ACC. WITH AN ARROW
0180  STA $9C40      ;SCREEN RAM
0190  JMP BEGIN      ;DO IT AGAIN
```

As long as comments are preceeded with a ";", a comment can contain anything, (letters, numbers, symbols,etc.) just like comments following a REM statement in BASIC. When the assembler converts the soure code to object code, the comments are ignored.

Psuedo Opcodes

You have probably also noticed that the first line of every assembly language program you have seen thus far contains an "*" followed by an "=" and an address (usually $0600). In assembly language you must tell the assembler where in memory to store the object code of your program. The Atari uses an asterisk to set the starting address of the program's object code in memory, which is referred to as the "origin" of the program. The equals sign is a "psuedo opcode." A psuedo opcode is an instruction to the assembler. For example, "*=$0600" instructs the assembler to set the origin of the program equal to $600. Psuedo opcodes are not translated into 6502 object code. They are instructions to the assembler. Turn to Assembly Language Programming Worksheet #4 to change the origin of the ARROW program.

Assembly Language Programming Worksheet #4

1. ENTER the ARROW program.

2. LIST the program and use the editing keys to move the cursor up over the first "0" in the address "$0600" on line 0100.

3. While holding down the CTRL key, press the DELETE key once. The DELETE key is in the upper right hand corner of the keyboard. The cursor should now be sitting over the "6" in "$600".

4. Now use the editing keys to move the cursor to the space just past the last "0" in "$600".

Type: 0 and press RETURN

The first line of your program should look like the following.

```
          0100  *=$6000
```

5. Press the BREAK key a few times to move the cursor down below the program. Now assemble the program.

6. Look closely at the addresses of the object code. They no longer start with 600. The object code is stored in memory starting at $6000 instead. And even though the first line of your program was "*=$6000", the first byte of the object code is A9, for the LDA instruction. The "*=" psuedo opcode is only an instruction to the assembler. The 6502 never processes it.

7. Go into the degugger to run the program. What instruction will you use to execute this program? (Hint: "G" stands for go, and the number which follows is the origin or starting address of the program in memory.)

Up to this point we have been storing the object code of the assembly language programs on page six of memory ($600-$6FF). Page six is a free area of RAM and a good place for short assembly language programs. As your programs get longer you can set the origin of your program to any address in the free RAM area between $2000-$A000. However, if you are using $9C40 - $A000 for screen RAM, as we are throughout this module, you should probably originate your program between $2000-$9000. Also, if you use the USR function to run an assembly language program from BASIC, you need to avoid having one program write over another in memory.

In assembly language it is possible to give a name to an address that you use in your program. For example, instead of using the address $9C40, we could assign the name SCREEN to the address. Then any time we wanted to store a value at that address, we could just use the name SCREEN. To assign a name to a variable or an address, we must use the "=" psuedo opcode.

Constant and variable declarations are grouped together in assembly language programs and commonly follow the origin statement at the beginning of the program. Take a look at the example below.

```
 *=$0600
SCREEN = $9C40     ;START GR.0 SCREEN
 LDA #$7D          ;LOAD ACC. WITH AN ARROW
 STA SCREEN        ;PUT A ON SCREEN
 RTS               ;RETURN
```

Note that the "S" in SCREEN is in the label field. All variable and constant declarations begin in the label field, one space before the command field.

As this program is expanded, any time you want to refer to the address, $9C40, you can just use the name SCREEN. Using constant and variable names in programs makes a program much easier to understand. Also, whenever you go to change the address you are using, all you need to do is change the constant declaration at the beginning of the program. From then on the assembler treats the word SCREEN as the new address. Otherwise, you need to search through your program to find every instance in which you used the address $9C40. As your assembly language programs get longer, locating all the instances of $9C40 becomes an extremely arduous task. To experiment with assigning a name to an address and then changing that address, turn to Assembly Language Programming Worksheet #5.

Assembly Language Programming Worksheet #5

1. ENTER the ARROW program on your advanced topics diskette.

```
Type: ENTER #D:ARROW  and press RETURN
```

2. LIST the program. To insert a line that assigns the name SCREEN to $9C40, we can just add another line number.

```
Type: 0105 SCREEN = $9C40  and press RETURN
          \     //
           Space
```

3. Now we need to replace the screen address in the STA instruction with the word SCREEN. Using the CTRL and the arrow keys, move the cursor up and place it over the $ in the STA $9C40 instruction.

```
Type: SCREEN  and press RETURN
```

4. Assemble the program, go into the debugger, and execute the program. Assigning a name to the screen address should not have affected the operation of your program in any way.

5. If you have difficulties assembling your edited version of the ARROW program, load the SCRADR program on your advanced topics diskette.

6. Experiment with changing the address of screen RAM you are using. The addresses for the screen range from $9C40 to $9FFF. Use the CTRL and arrow editing keys to put the cursor over the addresss in the SCREEN = $9C40 assignment. Change the address. Be sure to press RETURN after typing in a new address and move the cursor down below the program before trying to assemble it. Can you put the arrow in the middle of the screen?

For purposes of explanation, the address of screen RAM will be used instead of the name SCREEN in the next couple of programs.

Assembly Language Instruction Set And Beginning Addressing Modes

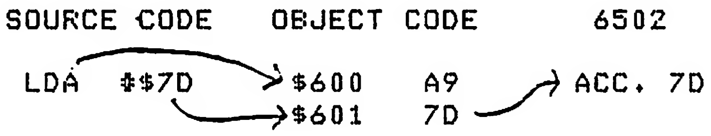
The most commonly used assembly language instructions will be explained and demonstrated in this section. Some of the addressing modes in assembly language also will be discussed.

There are 56 instructions in the Atari 6502 instruction set. Each instruction consists of a three letter mnemonic or an abbreviation of the operation the instruction performs.

The most common instructions are those that transfer data between the microprocessor and memory. All the data transfers that go on between the CPU and memory involve one of the internal registers. "Load" instructions transfer memory data into the accumulator, the X register, or the Y register. There is a set of three load instructions - one for each register.

LDA:	LoaD the Accumulator
LDX:	LoaD the X Register
LDY:	LoaD the Y Register

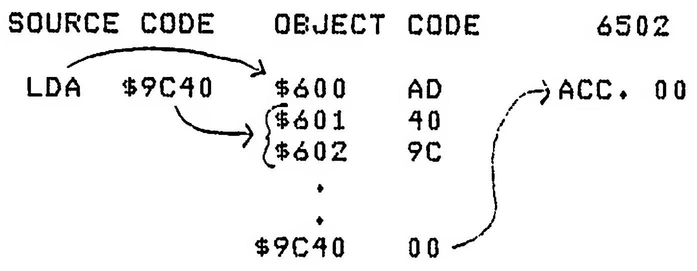
You are familiar with the LDA instruction.



```
SOURCE CODE   OBJECT CODE      6502

 LDA  #$7D     $600    A9      ACC. 7D
               $601    7D
```

The value immediately following the opcode for the LDA instruction in memory is stored in the accumulator. The "#" is referred to as an "immediate" symbol. So the LDA #$7D is read, "load the accumulator with immediate hexadecimal $7D." Whenever you use a hexadecimal number, you must precede the value with a "$". To use decimal numbers in a program, you simply forgo the dollar sign. LDA #125 is the same as LDA #$7D since decimal 125 equals hexadecimal $7D. The "#" remains because we are still loading the accumulator with the value immediately following the instruction. The load instructions for the X and Y registers function exactly the same way. LDX #$7D places hexadecimal $7D in the X register. LDY #$7D places a hexadecimal $7D in the Y register. Loading a register with a specific value is called "immediate addressing." Immediate addressing is easily recognized by the "#" preceding the value to be loaded into the register.

It is also possible to load a register with the contents of a memory location. Suppose you have a program that computes a math problem and stores the answer in memory. When the program is done, you don't know what the answer is, but you do know the memory address of where the answer has been stored. You need to be able to load a register with the contents of the address of the answer so you can find out what the answer is. Loading a register with the contents of a memory location is called "absolute addressing." In absolute addressing, the operand to the instruction is the address of the memory location you wish to see. Study the diagram below to see how absolute addressing works.



```
SOURCE CODE    OBJECT CODE      6502

 LDA  $9C40    $600    AD      ACC. 00
               $601    40
               $602    9C
                 .
                 .
              $9C40    00
```

The zero stored in $9C40 is loaded into the accumulator. Since this is absolute addressing, the "#" is no longer used. Note that the opcode for the LDA instruction stored in $600 is "AD". Up until now the opcode for LDA has been A9. The opcode changed because the operation performed by the CPU is different. AD instructs the CPU to get the value stored in the specified memory location and load it into the register. The AD also instructs the CPU to fetch three bytes, one byte for the opcode of the instruction, and two bytes for the address in the operand. You needn't worry about what the specific values are of the various opcodes, or which opcodes represent which addressing modes. The assembler and the processor handle that for you. Our goal here, is to point out that the opcode indicates to the CPU the type of addressing being used and thus, what operation the CPU is to perform.

Assembly Language Programming Worksheet #6

Turn off the computer and reboot your system to begin this worksheet. It is necessary for you to start with empty memory and empty registers.

1. ENTER and LIST the ARROW program.

2. Use the edit keys to move the cursor up over the LDA #$7D instruction. Change the instruction to read "load the accumulator with immediate decimal 64." What number will be stored in the accumulator? ________ Be sure to press RETURN after editing the LDA instruction.

Assemble the program, go into the debugger, and run the program (G600). When the program stops, the registers will be listed. Were you right?

3. Type X to go back to the editor and LIST the program. Now change the LDA #64 instruction to LDA #298. What will be loaded into the accumulator? ________ Assemble the program.

That was a trick question. You should have gotten Error 10. Page 43 of the Assembler Editor manual lists the error messages. Error 10 states, "the expression is greater than 255 where only one byte is required." Remember that one memory location holds a maximum of 255. If you try to load a number greater than 255 into the accumulator, the program will not assemble.

4. Now let's try some absolute addressing. LIST the program. Replace LDA #298 with LDA $600. What value will be loaded into the accumulator? ________ If you are unsure, assemble the program and then try to answer the question. The object code for the LDA instruction should appear as follows.

```
          0600  AD0006   0110    LDA $600
```

LDA $600 loads the accumulator with the contents of memory location 600. What is the value in $600 which will be loaded into the accumulator?

5. Run the program from the debugger and check the contents of the acuumulator against your answer.

6. Define the addressing modes used below and explain what the instruction will do.

LDA #$7D ---

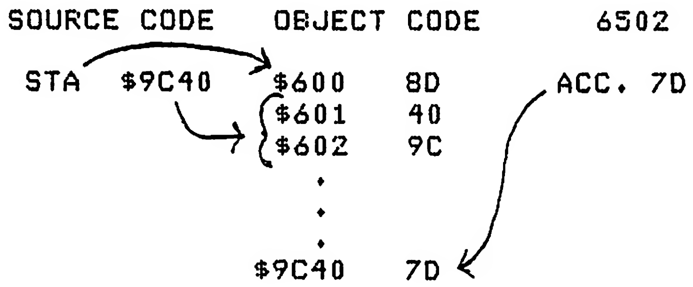
LDA #64 ---

LDA $9C40 ---

Whenever you want to put a value in memory, you use a "store" command. There are three store instructions, one for each register.

STA: STore the value in the Accumulator in memory.
STX: STore the value in the X register in memory.
STY: STore the value in the Y register in memory.

In the ARROW program the STA instruction was used to put the value for an arrow into memory location $9C40. (This is another example of absolute addressing.)



The $7D in the accumulator is stored into memory location $9C40. Actually, a copy of the $7D is made and stored in $9C40. The $7D in the accumulator remains unaffected by the STA command. Turn to Assembly Language Programming Worksheet #7 to try the different load and store instructions.

Assembly Language Programming Worksheet #7

You will need to turn off your machine and reboot the system with an Assembler Editor cartridge and the advanced topics diskette in order to insure that the registers are all empty when you begin this worksheet.

1. ENTER and LIST the ARROW program.

2. Use the editing keys to place the cursor over the "A" in the LDA instruction. Instead of loading the accumulator with #$7D, load the X register with #$7D. Type an "X" to replace the "A".

3. If the value for the arrow is being loaded into the X register, then to print the arrow on the screen, we must store the contents of the X register in screen RAM ($9C40). Change the STA command to a ST_X_ command.

4. Assemble the code. Type BUG to get into the debugger. Type SHIFT/CLEAR, to clear the screen so the arrow won't scroll up off the screen, and run the program from $600 by typing G600.

5. List the contents of the different registers below. The contents of the internal registers will be listed at the bottom of the screen once the program is completed.

A=   X=   Y=

As you can see, the program's performance does not change by using the load and store instructions for the X register. However, now the value for the arrow is stored in the X register instead of in the accumulator. Now let's investige where the #$7D ends up when the program is executed.

Type: D9C40 and press RETURN

The "D" stands for display. We are displaying the contents of memory location $9C40.

You should see a 7D. A copy of the 7D in the X register has been stored in $9C40.

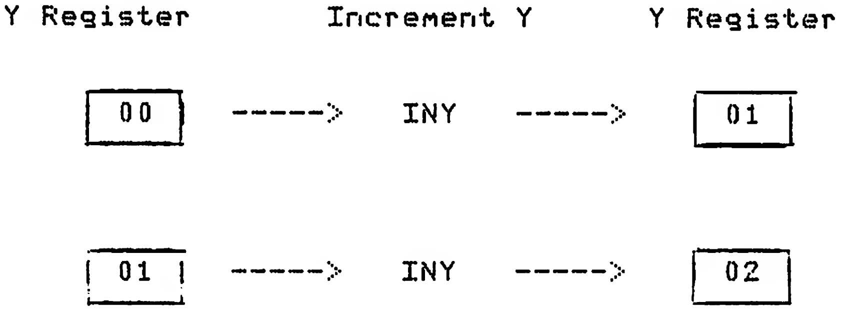
Let's get on with some assembly language programming. You saw that a short assembly language program, which places an arrow on the screen is executed so quickly that you can't even see the arrow displayed. The alternative program we have used leaves the character on the screen. What good is assembly language if we can't control how long something will be displayed on the screen? What we need is a "delay loop," which acts as a timer. Suppose we put the arrow on the screen and then we set a timer to count to 255. While the arrow is being displayed on the screen, the timer ticks away. When the timer gets to 255, the next instruction in the program is executed.

To simulate a timer (or write a delay loop) we need to use an "increment" instruction that adds one to a counter. There are three increment instructions.

INC: Add one to the contents of a memory location.
INX: Add one to the contents of the X register.
INY: Add one to the contents of the Y register.

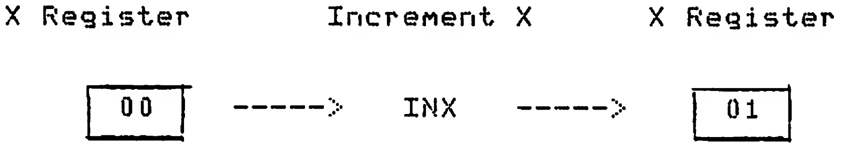
Note that there is no increment instruction for the accumulator. The INC instruction will be explained later.

The diagram below illustrates how the INY (INcrement the Y register)instruction works.



The 6502 handles the addition for you and stores the new value in the Y register.

The X register can be incremented in the same way with the INX instruction.



The INX and INY instructions are self-sufficient commands. There is no operand necessary for INY or INX. When an instruction contains all of the information the CPU needs, it is called "implied addressing." With the INY and INX instructions, the object of the operation is the register, which is implied by the instruction itself.

```
*=$0600
LDY #00        ;LOAD Y WITH 0
INY            ;ADD ONE TO THE VALUE IN Y
RTS            ;RETURN
```

RTS is another example of an instruction that uses implied addressing. It does not require an operand. The CPU understands from the RTS instruction alone that it is to return to BASIC or to the program that called the routine.

It is not possible to increment the accumulator. Instead, the third increment instruction enables you to add one to the contents of a memory location. For example, suppose you have a variable called "COUNTER" in your program and it is stored in memory location $CD. ($CD is a free memory location on the zero page of memory.) Look over the program below.

```
 *=$0600
COUNTER = $CD      ;ASSIGN COUNTER TO $CD
 LDA #00           ;LOAD ACC. WITH 0
 STA COUNTER       ;INITIALIZE COUNTER
 INC COUNTER       ;ADD ONE TO THE VALUE IN COUNTER
 RTS               ;RETURN
```

COUNTER is initially set to 0. When the INC COUNTER instruction is executed, one is added to the value stored in COUNTER. It is also possible to place an actual address in the operand of an INC instruction. For example, in the program above, INC $CD would have served the same function as INC COUNTER. However, using variable names is highly recommended. Variable names make programs more understandable both to the programmer and anyone else reading the program. Variable names also enable you to easily alter or update a program. To experiment with the increment commands turn to Assembly Language Programming Worksheet #8.

Assembly Language Programming Worksheet #8

To begin this worksheet, you will need to turn off your machine and reboot the system with an Assembler Editor Cartridge and the advanced topics diskette in order to insure that the registers are all empty.

1. You should have the EDIT prompt on the screen. Type in the following program. Be sure to leave two spaces between the line number and the instruction for the label field. Press RETURN after entering each line.

```
100  *=$600
110  LDY #$A0
120  INY
130  RTS
```

2. After running this program, what number would you expect to find in the Y register?_______ Execute the program from the debugger and see.

3. To get back to the editor:

   Type: X   and press RETURN

4. LIST the program. Using the editing keys, place the cursor over the value being loaded into the Y register ($A0). Replace the number with the values listed below. Fill in the boxes with the new values held in the Y register after executing the program.

Y Register				Y Register
09	----->	INY	----->	
FE	----->	INY	----->	
FF	----->	INY	----->	

When you incremented $FF, you should have gotten 00 in the Y register. $FF is the largest two digit hexadecimal number. When one is added to $FF, the sum is $100.

```
$ FF
 +01
$100
```

Similarly, in base 10 (decimal), 99 is the largest two digit number that can be represented. Adding one to 99 resets the two digits to 0 and carries a one over into the next place value. Since registers and memory locations in the Atari only hold one byte, when one is added to $FF, the Y register is reset to zero.

5. In order to have a look at the contents of the Y resgister, step through the last program, which increments $FF. From the debugger,

Type: S600   and press RETURN

First, the LDY #$FF instruction is executed and the Y register is set to $FF.

Type: S   and press RETURN

This time the INY instruction is executed. At the bottom of the screen you should see the following. (Don't worry if the S for stack pointer does not equal 08.)

```
0602        C8          INY
     A=00    X=00    Y=00    P=32    S=08
```

The "P=" stands for the processor status register. The status register is one of the internal registers in the 6502. The status register holds one byte, however, each bit holds significant information concerning the results of the CPU's most recently executed instruction. For example, if the last instruction left a negative number in one of the registers, the negative bit of the status register would be set. (The status register was first introduced in the Machine Architecture Module. See the Central Processing Unit section if you would like a review.) Each bit of the status register is called a flag and it indicates if a certain condition exists in the processor. Currently, the status register on your screen should hold a 32 (P=32). The binary representation of the status register below shows the bit pattern for the hexadecimal number $32. The ones indicate which bits of the status register are set.

Status Register

```
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |

  N   V   -   B   D   I   Z   C
                          ↑
```

The "Z" bit, or zero flag, is set. The result of the last instruction (INY) left a zero in the Y register, and consequently the zero flag of the status register was set. (The "-" or unused bit and the "B" or the break bit were also set. These flags remain set as a program is executed. You needn't worry about why they are set.) The importance of the zero flag will become clearer in the next section. Don't worry if you are confused by the status register flags. They are typically difficult for beginners to understand. The status flags will become clearer the more you program in assembly language.

Assembly Language Programming Worksheet #9

There is also a set of "decrement" instructions, which are the opposite of increment instructions.

DEX subtracts one from the value in the X register.

DEY ______________ one from the value in the ___ register.

DEC subtracts one from the contents of a memory location. DEC COUNTER subtracts one from the value stored in COUNTER.

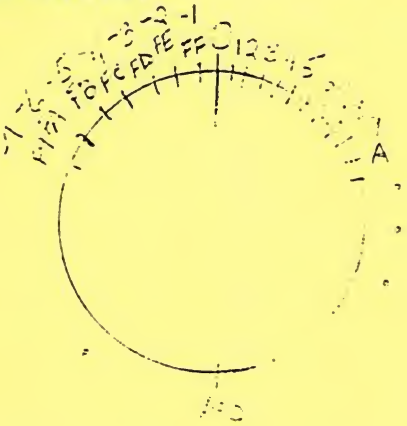
1. Use the editing keys to change the increment command in the increment routine, which you used in worksheet #8, to a decrement instruction as listed below. If you no longer have the increment program in memory, type in this new program.

```
100  *=$600
110  LDY #$FF
120  DEY
130  RTS
```

Assemble the program and run it from the debugger. Try the different values for the Y register listed below. Fill in the boxes for the result of the DEY instructions.

Y Register				Y Register
FF	---->	DEY	---->	□
F0	---->	DEY	---->	□
00	---->	DEY	---->	□

Decrementing 00 should have given you $FF in the Y register. In assembly language $FF stands for a minus one as well as 255. The CPU uses a circular number line. Take a look at the diagram below.



If you add $FF to 0, you get $FF. If you subtract one from zero, you also get $FF. The processor knows whether $FF represents a minus one or 255 according to the status register flags. When one is subtracted from 00, the result is $FF and the negative bit of the status register is set. When 255 is added to zero, none of the significant status flags are set.

2. Step through the last decrement program, which subtracts one from zero. Type S600. The contents of the registers will be listed as each instruction is executed. The Y register should hold 00, from the LDY #$00 instruction.

3. Type S to execute the DEY instruction, and press RETURN. The current contents of the registers will be listed. Fill in the registers below with what appears on your screen.

```
0602     88     DEY
      A=     X=     Y=     P=     S=
```

4. The status register (P) should have "B0" in it after executing the DEY instruction. Remember that the status register holds the status flags. Each bit of the status register holds significant information. The binary bit pattern for B0 and the status flags associated with each bit are shown below.

Status Register

1	0	1	1	0	0	0	0
N	V	-	B	D	I	Z	C
↑							

The "N" or negative flag has been set to indicate that decrementing 00 resulted in a negative number.

Don't worry if you don't understand the peculiar numbering system of the CPU.

To set up a loop that repeatedly increments or decrements a register (or a counter), we need to use a "branch" instruction. The "BNE" instruction stands for Branch Not Equal to zero. BNE can be used to repeat a decrement instruction until the register has reached zero. Take a look at the short program below which uses a BNE instruction for a timing loop.

```
 *=$600
SCREEN = $9C40
 LDY #$FF      ;SET COUNTER
 LDA #$7D      ;CODE FOR AN ARROW
 STA SCREEN    ;DISPLAY
DELAY DEY      ;SUBTRACT 1 FROM Y
 BNE DELAY     ;IF Y IS NOT 0,REPEAT DELAY
 RTS           ;RETURN
```

In the example above, as long as the Y register is not zero, the CPU will branch back to the label "DELAY" and decrement the Y register again.

To determine if the Y register has reached zero, the BNE instruction checks the zero flag of the status register. When the register is decremented to zero, the zero flag of the status register is set. When the BNE instruction finds that the zero flag of the status register is set, the condition for branching when the register is not equal to zero is no longer in effect. The register is zero and so the branch is not taken. Instead, the next instruction in the program is executed.

The 6502 instruction set has a series of branch instructions, each of which checks the current condition of one of the status flags. You can branch on a negative number, a positive number, a carry, etc. Below are the eight branch instructions available with the Atari assembler editor.

BCC:	Branch on Carry Clear
BCS:	Branch on Carry Set
BEQ:	Branch on EQual to zero
BMI:	Branch on result MInus
BNE:	Branch Not Equal to zero
BPL:	Branch on result PLus
BVC:	Branch on oVerflow Clear
BVS:	Branch on oVerflow Set

Branch instructions are very useful for short distance branches, as is the case with timing loops. However, it is not possible to branch long distances in a program. In a large program where a long branch is needed, the alternative to a branch instruction is a "JSR", jump to a subroutine. JSR will be explained in the next section.

Turn to Assembly Language Programming Worksheet #10 to see how a delay loop works in the ARROW program.

Assembly Language Programming Worksheet #10

1. ENTER the HOLDARROW program on your advanced topics diskette.

Type: ENTER #D:HOLDARROW and press RETURN

```
0000          0100          *=     $0600
9C40          0110 SCREEN =        $9C40
0600 A000     0120          LDY #$00      ;SET COUNTER
0602 A97D     0130          LDA #$7D      ;ARROW CODE
0604 8D409C   0140          STA SCREEN    ;DISPLAY
0607 C8       0150 DELAY    INY           ;ADD 1 TO COUNTER
0608 D0FD     0160          BNE DELAY     ;IF NOT 0, REPEAT DELAY
060A 60       0170          RTS           ;RETURN
```

2. LIST the program. It should look like the listing above. The Y register serves as a timer which counts to 255 while the arrow is being displayed on the screen.

3. Assemble the program and execute it from the debugger. You would think that because the computer has to count to 255, the arrow would stay on the screen longer before the RTS forces the screen to scroll up. It doesn't look much different does it? It is longer, though. Step through the program to see that the Y register is really being incremented 255 times while the arrow is on the screen. Do the following.

Type: S600 and press RETURN

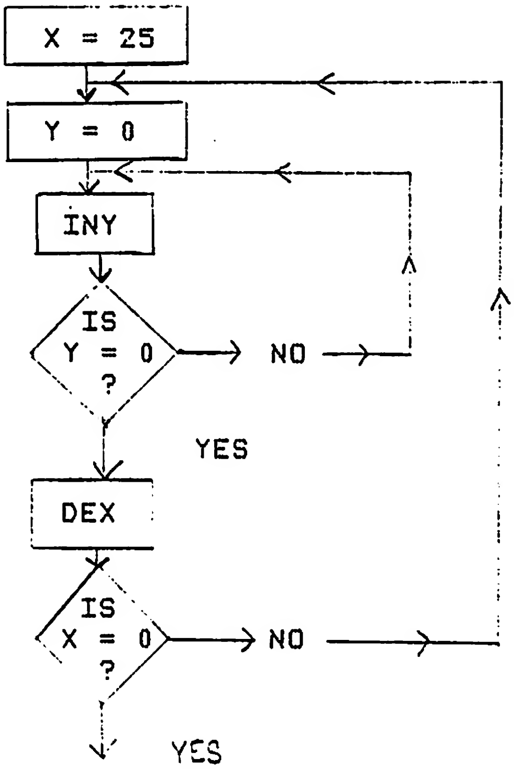
Continue to type "S" and RETURN a few times to see the Y register being incremented.

The branch instruction is always followed by a label to an instruction which is close by in the program. There must be a short distance between the instructions because branch instructions use "relative addressing." The object code for a branch command is two bytes, one byte for the instruction, and one byte for the "offset," or the distance of the branch. The offset is the number of bytes in memory between the branch instruction and the instruction accompanying the label you are branching to. Look at the object code for the branch command in the HOLDARROW program listed below.

```
0607 C8      0150 DELAY  INY            ;ADD 1 TO COUNTER
0608 D0FD    0160        BNE DELAY      ;IF NOT 0, REPEAT DELAY
```

Memory location $608 holds, D0, the opcode for the BNE instruction. The FD in $609 is the offset to the label DELAY. FD, in this case, represents a decimal -3. The CPU must look back three bytes in memory to find the instruction associated with the label DELAY. Since the offset is one byte in the object code, the distance that is branched must be held in one byte. Consequently, you can branch up to 128 bytes forward ($00-$80), and 127 bytes back ($81-$FF) in a program and no further. Branch instructions are the only assembly language instructions that use relative addressing. The offsets in the object code are handled by the CPU. All you need to worry about is branching too far in your programs.

A longer delay is needed in order to leave the arrow on the screen for a longer period of time. To create a longer delay we will need to use another register. This second register will count the number of times the first register counts from 0 to 255. What we will do is "nest" the 0-255 timing loop inside another loop. Suppose we load the X register with 25 and each time the Y register counts from 0 - 255 the X register is decremented. This cycle is continued until the X register is zero.



Here is the assembly language version of the nested delay loops illustrated in the flow chart.

```
DELAY LDX #25     ;COUNTER FOR Y LOOPS
AGAIN LDY #00     ;0-255 COUNT
 WAIT INY         ;ADD 1 TO Y
      BNE WAIT    ;IF NOT 0, REPEAT WAIT
      DEX         ;SUBTRACT 1 FROM X
      BNE AGAIN   ;IF NOT 0, REPEAT AGAIN
      RTS         ;RETURN
```

The delay loop is now a separate subroutine, which the ARROW routine will "call." The advantage of making the delay loop a separate subroutine is that it can be used from any-where in an assembly language program. As you have seen, assembly language is processed so rapidly that delay loops are commonly needed. If the nested delay loop had been incorporated into the ARROW program, it could only be used when a character was being printed in the upper left hand corner of the screen. The secret to good assembly language programming is to write versatile subroutines that can be reused within the program.

Turn to Assembly Language Programming Worksheet #11 to experiment with changing the length of the delay.

Assembly Language Programming Worksheet #11

1. ENTER the SUBROUTINE program on the advanced topics diskette.

Type: ENTER #D: SUBROUTINE and press RETURN

The listing of the program should look like this:

```
          *=$600
         SCREEN = $9C40
          LDA #$7D         ;CODE FOR CLOVER
          STA SCREEN      ;DISPLAY
          JSR DELAY       ;WAIT
          RTS             ;RETURN
;
;
;
         DELAY LDX #$A0    ;COUNTER FOR Y LOOPS
         AGAIN LDY #00    ;0-255 COUNT
         WAIT INY         ;ADD 1 to Y
          BNE WAIT        ;IF NOT 0,REPEAT WAIT
          DEX             ;SUBTRACT 1 FROM X
          BNE AGAIN       ;IF NOT 0, REPEAT AGAIN
          RTS             ;RETURN
```

The "JSR" instruction, which stands for <u>J</u>ump to the <u>S</u>ub<u>R</u>outine, is used to call the delay routine. The RTS instruction at the end of the delay routine tells the CPU to go back to executing the instructions in the ARROW routine.

2. The value stored in the X register controls the length of the delay. Assemble the program and execute it from the debugger to see how long the delay lasts.

3. To return to the editor,

Type: X and press RETURN

4. Replace the #$A0 in the LDX #$A0 command with #$F0. Assemble and run the program from the debugger. What effect did changing the value in the X register have on the delay?

----

5. What would happen it you changed the value loaded into the X register to #$5?

Try it and see.

Summary

The 6502 offers eight different addressing modes. The addresssing modes that have been covered thus far are listed below.

Immediate	LDA #$7D
Absolute	STA $9C40
Implied	INX, RTS
Relative	BNE AGAIN
Zero Page	LDA $CD

Zero page addressing is the same as absolute addressing, except that the address being accessed is on the zero page. Addresses on the zero page are listed as one byte because the high order byte of the address is "00". The complete address of $CD is $00CD. When zero page addressing is used, the object code for the command is only two bytes, one byte for the instruction, and one byte for the address. The CPU assumes that the high order byte of the zero page address is $00. Variables that are used frequently in a program are commonly stored on the zero page for quick and easy access.

Indexed Addressing Modes

This section covers the three remaining addressing modes used in 6502 assembly language. Two of the three indexed modes will be used in the final animation program.

How about printing something a little more interesting than an arrow on the screen. Suppose you wanted to print four lines in succession, which would look like a stick spinning or a pinwheel. Four lines available in the internal character set are listed below.

<table>
<tr><th></th><th></th><th>HEX</th><th>DECIMAL</th></tr>
<tr><td>|</td><td>=</td><td>$7C</td><td>124</td></tr>
<tr><td>/</td><td>=</td><td>$0F</td><td>15</td></tr>
<tr><td>-</td><td>=</td><td>$0D</td><td>13</td></tr>
<tr><td>\</td><td>=</td><td>$3C</td><td>60</td></tr>
</table>

One possiblity is to repeatedly load the accumulator with the values for each of the four lines. For example, we could write the following program.

```
 *=$600
SCREEN = $9C40
 LDA #$7C          ;CODE FOR |
 STA SCREEN        ;DISPLAY
 LDA #$0F          ;CODE FOR /
 STA SCREEN        ;DISPLAY
 LDA #$0D          ;CODE FOR -
 STA SCREEN        ;DISPLAY
 LDA #$3C          ;CODE FOR \
 RTS               ;RETURN
```

It works, but this certainly is an inefficient way of going about printing a pinwheel. Instead, it would be preferable to have one set of instructions that printed a line on the screen. The code for the different lines would be passed through the printing routine. This would eliminate the repetition of LDA and STA instructions. In assembly language it is possible to set up a data table and read through the data one element at a time, just the way you can in BASIC.

To store the codes for these lines as data in memory, the psuedo opcode ".BYTE" can be used. The .BYTE command informs the assembler that what follows is a series of bytes which are to be stored in successive memory locations. Not every assembler uses the .BYTE command. Some assemblers have different psuedo opcodes for saving data. To use the .BYTE command, the data must be listed in decimal and separated by commas. The .BYTE command that holds the data for the four lines is listed below.

```
Label Instruction
   /    \
 CHAR .BYTE  124,15,13,60
```

CHAR is the label used to identify where the data is stored in memory. The data are listed in the operand of the command field. Each number in the list of data must be equal to or less than 255, since each element of data is stored in one memory location. When the assembler converts the source code to object code, an address is assigned to the label CHAR. If the address of CHAR is $060E, then the first element of data following .BYTE will be stored in $060E. The second element of data will be stored in $060F and so on.

Address	Data
$060E	$7C
$060F	$0F
$0610	$0D
$0611	$3C

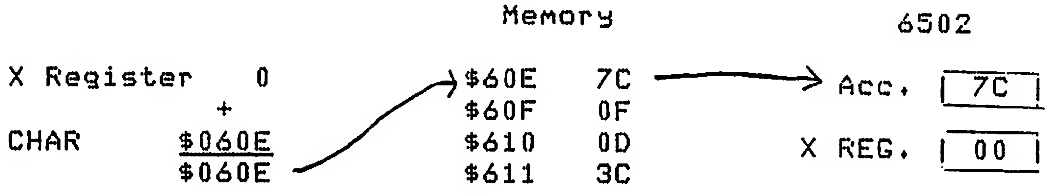
Now that the data are stored in memory, we need to be able to get the numbers to be printed on the screen, one at a time. Reading through data in assembly language is accomplished with "indexed addressing." The X register or the Y register serves as an "index" for reading through the data. The following format is used for indexed addressing.

```
LDA CHAR,X
```

The number in the X register is added to the address of CHAR. The value in this new address is loaded into the accumulator. For example, suppose the X register contains a zero.

```
LDA CHAR,X
      /    \
  $060E + 0 = $060E
```

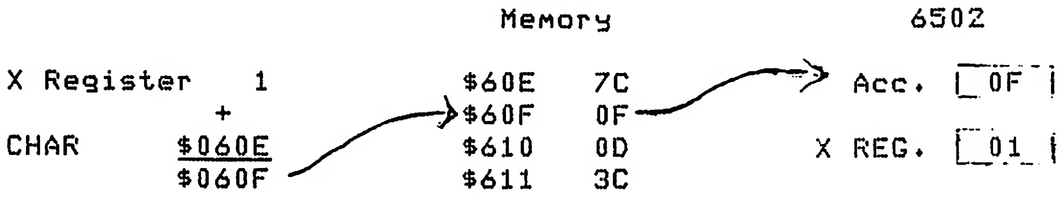
Zero is added to $060E, the address of CHAR. The accumulator is loaded with the contents of this new address.



The first byte of data is stored in the accumulator. Suppose we incremented the X register to one.

```
       LDA CHAR,X
             /   \
        $060E + 1 = $060F
```

This time the value in $060F is loaded into the accumulator.



Either the X register or the Y register can be used as an index. With indexed addressing you can access any one of 255 elements of data stored in memory. You are restricted to a maximum index of 255, because that is the largest number the X or the Y register can hold. Turn to Assembly Language Programming Worksheet #12 to see how you can incorporate indexed addressing and the .BYTE psuedo opcode into your assembly language programs.

Assembly Language Programming Worksheet #12

1. ENTER and assemble the PINWHEEL program on the advanced topics diskette. The listing on your screen should match the listing below. (The first line will not show.)

```
0000          0100           *=$600           ;ORGIN
9C40          0110   SCREEN = $9C40           ;SCREEN RAM
0600 A200     0120           LDX #$00         ;SET INDEX TO 0
0602 BD0E06   0130   NEXTCHAR LDA CHAR,X      ;GET NEXT CHAR
0605 8D409C   0140           STA SCREEN       ;DISPLAY IT
0608 E8       0150           INX              ;ADD ONE TO INDEX
0609 E004     0160           CPX #$4          ;COMPARE X REG. TO 4
060B D0F5     0170           BNE NEXTCHAR     ;IF X <> 4 BRANCH
060D 60       0180           RTS              ;RETURN
060E 7C       0190          CHAR .BYTE 124,15,13,60  ;DATA
060F 0F
0610 0D
0611 3C
```

2. Have a look at the object code.

3. What is the opcode for the LDA in the CHAR,X instruction?_______ Another opcode for the LDA instruction! "BD" instructs the processor to take the contents of the X register, add it to the address of CHAR, and store the contents of the new address in the accumulator. (The opcode also tells the CPU to fetch two bytes in the operand following the opcode BD. The two bytes following the BD in the object code are the address of CHAR.)

4. Now look down at the contents of $060E - $0611. These are the bytes of data for the four lines that make the pinwheel. Note that there is no opcode for the .BYTE instruction. Psuedo opcodes are instructions to the assembler. They are not processed by the CPU. Also note that the .BYTE instruction and the pinwheel data are listed in the program following the RTS instruction. The data table must follow the RTS, because the data does not contain an instrucion or opcode for the CPU to execute. If the data came before the RTS, the CPU would try to interpret the data as opcodes to be executed.

5. A new instruction appears on line 160. "CPX" is one of a series of "compare" instructions.

CMP: CoMPare Memory and the Accumulator
CPX: ComPare Memory and the X Register
CPY: ComPare Memory and the Y Register

The branch instructions we used earlier in this module branched until either 0 or 255 was reached. Compare instructions enable the programmer to devise a loop with a termination point other than 0 or 255. CPX compares the contents of the X register with the number in the operand of the compare instruction. CPX #$4 compares the contents of the X register with 4. The comparison is made by subtracting the operand, 4, from the value held in the X register. In the PINWHEEL program the X register is increemented just prior to the compare instruction. So the first time the CPX #4 is executed, the X register is one.

CPX #$4

```
 01   X Register
-04   Compare Operand
 -3
```

The answer, -3, sets the negative bit of the status register. Compare instructions set the negative, zero, or carry bit of the status register, depending on the results of the subtraction. There is no other evidence of the subtraction or execution of the compare instruction. The number in the X register remains the same as it was prior to the compare instruction. When the X register is incremented to four and compared to the 4 in the CPX instruction, the result of the comparison is zero.

CPX #4

```
 04   X Register
-04   Compare Operand
 00
```

The result of the comparison will set the zero flag of the status register. In the PINWHEEL program a BNE instruction is used to check the zero flag of the status register. Thus, the first through the fourth elements of data will be loaded into the accumulator and stored on the screen with indexed addressing. When the X register is incremented to 4, the BNE (branch not equal to zero) is no longer effective. The zero bit has been set, so the branch is not taken, and the next instruction in the program is executed.

6. Finally, let's run the program.

Type: BUG   RETURN   G600

According to the way we have planned the program, you should see the four lines displayed, one right after the other, giving the appearance of one spin of a pinwheel. However, all we see is one line. We are up against a speed problem again. The computer is processing the program and displaying the lines so fast that all we can see is the last line. To be sure that each of the four lines is being printed, replace the RTS instruction at the end of the program with a jump back to the beginning of the program. Use the CTRL and arrow keys to place the cursor over the "R" in RTS.

Type: JMP BEGIN and press RETURN

The JMP instruction is similar to a GOTO in BASIC.

To insert the label BEGIN, place the cursor over the space before the LDX #$00 instruction. Hold down the CTRL key and press the INSERT key (in the upper right hand corner of the keyboard) five times - once for each letter in the word BEGIN.

Type: BEGIN and press RETURN

After you have typed BEGIN, be sure that there is a space in between the label BEGIN and the command LDX. Using the CTRL and arrow keys again, move the cursor down below the program.

7. Assemble the program and execute it from $600. At least we now know that each of the four lines is being stored in screen RAM as we intended.

To make the pinwheel look more like it is spinning, we need a brief delay after displaying each line. Ideally, we would simply insert a JSR DELAY into the routine that draws the pinwheel. However, we must first review how each of the subroutines is using the registers. It may be that one subroutine changes a register and affects the operation of the second routine. Look over the listing below. Focus on the use of the X register.

```
 *=$600          ;ORIGIN
SCREEN = $9C40 ;SCREEN RAM
;
DRAW LDX #$00   ;SET INDEX TO 0
NEXTCHAR LDA,X ;GET NEXT CHAR
 STA SCREEN     ;DISPLAY IT
 JSR DELAY      ;CALL DELAY ROUTINE
 INX            ;ADD 1 TO INDEX
 CPX #$4        ;COMPARE X REG. TO 4
 BNE NEXTCHAR   ;IF X=4 THEN BRANCH FOR CHAR
 RTS            ;RETURN
CHAR .BYTE 124,15,13,60  ;PINWHEEL
;
;
DELAY LDX #$A0 ;COUNTER FOR Y LOOPS
AGAIN LDY #$00 ;0 - $FF COUNTER
WAIT INY       ;ADD 1 TO Y
 BNE WAIT       ;IF NOT 0, REPEAT WAIT
 DEX            ;SUBTRACT 1 FROM X
 BNE AGAIN      ;IF NOT 0, REPEAT AGAIN
 RTS            ;RETURN
```

The X register is used as an index to CHAR and as a counter in the DELAY loop. The DRAW routine sets the X register to zero and loads the accumulator with the character to be printed on the screen. Then a delay is needed, so we JSR DELAY. In the course of the DELAY loop, both the X and the Y registers are manipulated. However, they are both at zero when the subroutine is completed. Thus, there is no conflict in the use of the X register the first time through the program. However, the Draw routine gradually increments the X register to index the line data. Suppose the X register has been incremented to one. When the DELAY loop is called, the X register is reset to zero. Immediately following the DELAY routine, the DRAW routine increments X. Consequently, the index to the data will be continuously reset to zero by DELAY and incremented to one in the DRAW routine. Since the X register would never get to four, the program would branch continuously and never stop. Thus, we need some way to preserve the index that reads through the data.

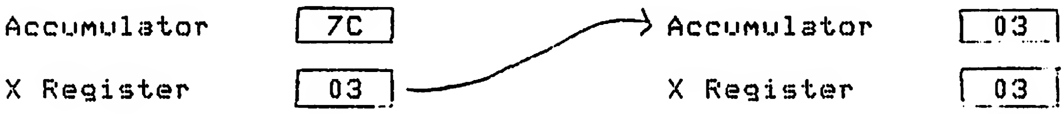
This is a good opportunity to employ the "stack," an area of memory reserved for temporary storage of information. Before calling the DELAY routine, we will save the current value of the index on the stack.

In the Machine Architecture module the "PHA" and "PLA" instructions were introduced. PHA stands for PusH the Accumulator onto the stack. PLA, PuLl the Accumulator off the stack, is used to retrieve the value from the stack. Any value to be put on the stack must first be put in the accumulator. So in order to save the X register on the stack, first we need to put the value in the X register into the accumulator. To shift a value from one register to another, we need to use one of a set of "transfer" instructions.

TXA: Transfer the contents of the X register to the Accumulator.
TAX: Transfer the contents of the Accumulator to the X register.
TYA: Transfer the contents of the Y register to the Accumulator.
TAY: Transfer the contents of the Accumulator to the Y register.

Transfer instructions make a copy of the value in one register and store that value in another register, as shown below.

TXA

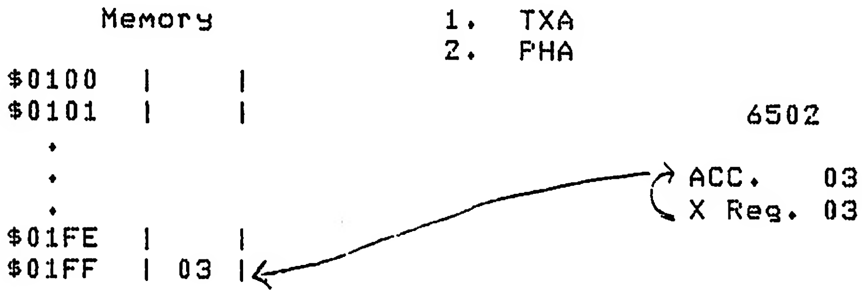


A copy of the contents of the X register is put in the accumulator. The X register remains intact.

None of the transfer instructions require an operand. All of the information the CPU needs is evident from the instruction, so implied addressing is used. Glance over the use of the PHA, PLA, and the transfer instructions below.

```
          TXA                  ;TRANSFER X INDEX TO ACCUMULATOR
          PHA                  ;SAVE IT ON THE STACK
          JSR DELAY            ;CALL DELAY LOOP
          PLA                  ;RETRIEVE INDEX FROM STACK TO THE
                               ;ACCUMULATOR
          TAX                  ;TRANSFER INDEX FROM ACCUMULATOR TO X
                               ;REGISTER
```

The index in the X register is transferred to the accumulator. PHA pushes the index, which is now in the accumulator, onto the stack. (The stack fills from $01FF down to $0100.)



The JSR DELAY sends the CPU to DELAY to execute the subroutine. When the delay loop is completed, it returns the CPU to the instruction following the JSR DELAY in the DRAW routine. PLA retrieves the index from the stack and puts it into the accumulator. TAX transfers the index, in the accumulator, back to the X register. Turn to Assembly Language Programming Worksheet #13 to see how this sequence of instructions has been incorporated into the DRAW routine. This time the pinwheel will spin.

Assembly Language Programming Worksheet #13

1. ENTER the SPIN program on your advanced topics diskette.

```
Type:  ENTER #D:SPIN
```

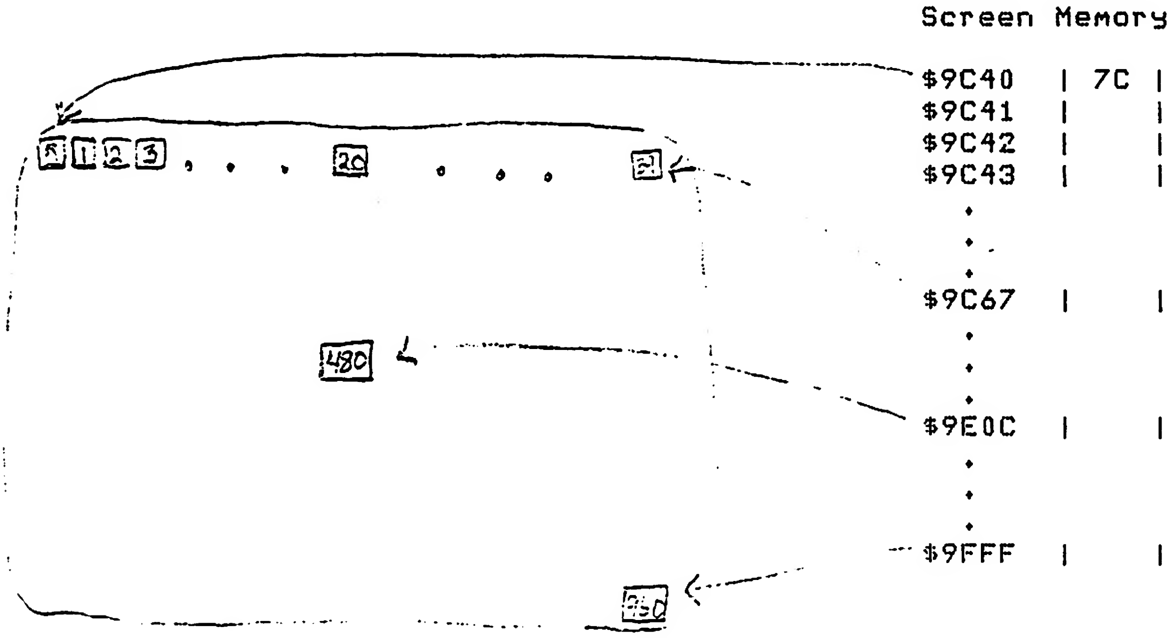
2. LIST lines 150 to 250 to see how the transfer commands have been incorporated into the DRAW routine.

3. Assemble the program and execute it from the debugger.

4. We can transfer the accumulator to the X register, and the X register to the accumulator. The Y register also can be transferred to the accumulator and vice versa. However, there is no instruction for transferring data between the X and Y registers. How can you transfer the X register to the Y register using the transfer commands you have learned? Write the assembly langauge code below.

```
     Command                  Comments

------------------        ----------------

------------------        ----------------

------------------        ----------------

------------------        ----------------
```

Spinning the pinwheel in the corner of the screen is fun, but how about putting that pinwheel somewhere else on the screen? The graphics zero screen has 960 locations, and so there are 960 memory locations reserved, each of which correspond to one location on the screen. Up until now, we have been using $9C40, the "starting location" of the graphics zero screen. There are 40 locations per line and 24 lines on the graphics zero screen. If you multiply 40 by 24, you come up with the 960 locations on the screen mentioned earlier. The 40 locations on the top row of the screen are numbered from 0 to 39 in decimal, and correspond to memory locations $9C40 - $9C67. The second row is numbered 40-79. The corresponding addresses are $9C68 - $9C8F. The address of the middle of the screen is $9E0C, and the contents of the last location on the graphics zero screen is stored at $9FFF.

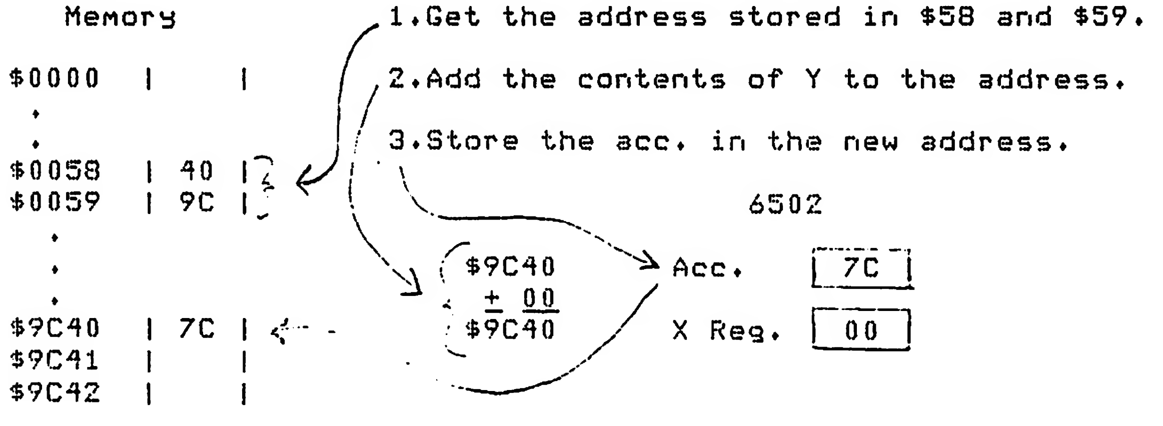


In order to move the pinwheel around on the screen, we need to be able to access any one of the 960 addresses ($9C40 - $9FFF) in screen RAM. One solution is to use "indirect indexed addressing." Indirect indexed addressing requires that the address to be indexed is stored on the zero page of memory. Quite conveniently, the starting address of screen RAM is stored in $58 and $59 on the zero page. (Memory locations $58 and $59 hold the starting address of the current graphics mode in use. See the Internal Representation of Graphics and Text module for an explanation of how the different graphics modes use memory.) For our present purposes $9C40 is stored in $58 and $59 on the zero page. The low order byte of the address, 40, is stored in $58. The high order byte of the address is stored in $59.

Indirect indexed addressing uses the Y register as an index. An example of an indirect indexed instruction is listed below.

```
STA ($58),Y
```

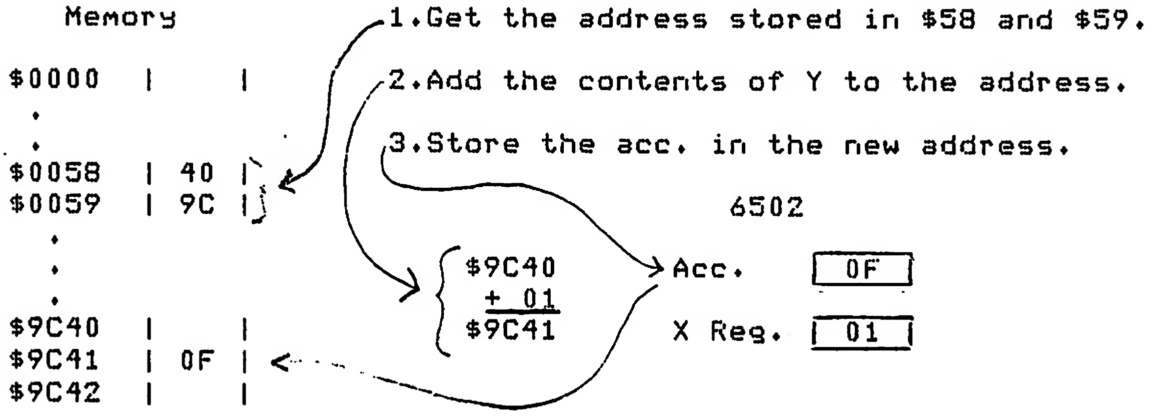
First, the CPU gets the addresss contained in $58 and $59. When the CPU encounters an opcode for indirect indexed addressing, it automatically takes the low byte of the zero page address given in the instruction and looks for the high order byte of the address in the next memory location. The value in the Y register is added to the address. The STA instruction then stores the value in the accumulator into the new address. Look over the diagram of the STA ($58),Y command below.


1.Get the address stored in $58 and $59.

2.Add the contents of Y to the address.

3.Store the acc. in the new address.

The STA instruction stores the accumulator in $9C40. Suppose the value in the Y register were incremented to one. To execute the STA ($58),Y instruction, first the CPU would fetch the address stored in $58 and $59. In our example the address is $9C40. Then one, from the Y register, is added to the address. The STA instruction uses this final address to store the accumulator in memory. Look over the diagram below.


Memory
1.Get the address stored in $58 and $59.
2.Add the contents of Y to the address.
3.Store the acc. in the new address.
$0000
$0058 | 40 |
$0059 | 9C |
6502
$9C40
+ 01
$9C41
Acc. 0F
X Reg. 01
$9C40
$9C41 | 0F |
$9C42

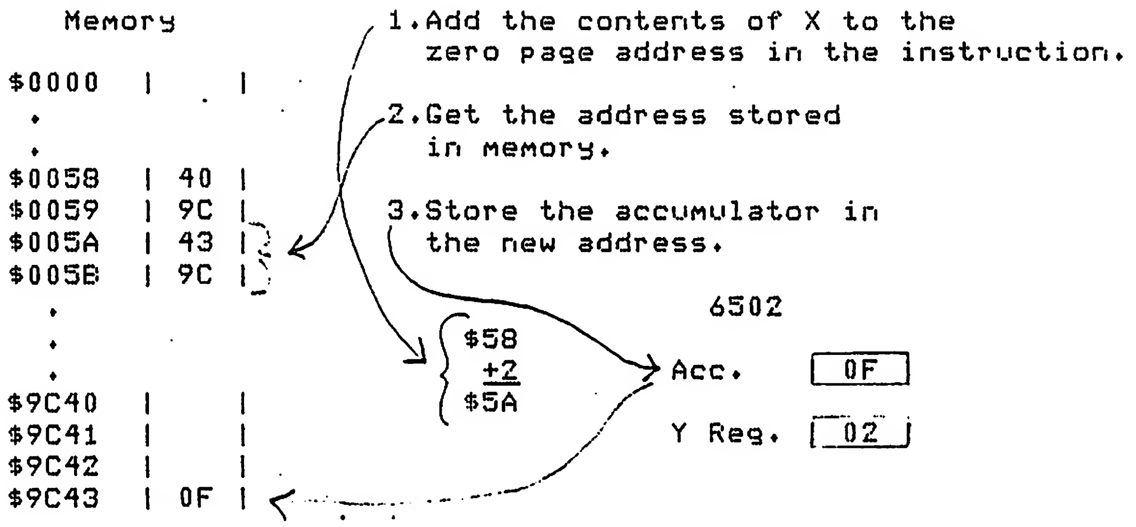
The address stored in $58 and $59 has not been changed. (In the programs that follow, the names LOWSCR and HISCR have been assigned to $58 and $59, because they hold the low byte and the high byte of screen RAM.

This is fairly difficult to understand at first. Don't panic. As you start programming in assembly language, you will see more applications for indirect indexed addressing, and it will become easier to understand.

There is one remaining 6502 addressing mode, which will not be used in the final animation program. "Indexed indirect" addressing is one of the least common addressing modes in assembly language. Only the X register can be used as an index in indexed indirect addressing. An instruction using indexed indirect addressing looks like the this:

```
                    STA ($58,X)
```

The value in the X register is added to the zero page address in parentheses. This new address contains another address. The accumulator is stored in this last address. Suppose the X register is 2 and the CPU is executing a STA ($58,X) instruction.



```
 Memory

$0000  |    |
  .
  .
$0058  | 40 |
$0059  | 9C |
$005A  | 43 |
$005B  | 9C |
   .
   .
   .
$9C40  |    |
$9C41  |    |
$9C42  |    |
$9C43  | 0F |
```

1.Add the contents of X to the zero page address in the instruction.

2.Get the address stored in memory.

3.Store the accumulator in the new address.

Thus, the value in the X register is added to the zero page address in order to get another memory address. Indexed indirect addressing is useful when you wish to access a certain element of data from various equal sized data tables stored in memory. You needn't worry if you don't understand the indexed indirect addressing mode just yet.

Animation

In this section you will write the assembly language routines necessary to move the pinwheel around on the screen. You also will learn how to read joystick data and move the pinwheel in the direction the joystick has been pushed.

First let's start by moving the pinwheel to the right across the screen. To move the pinwheel to the right, we need to add one to the pinwheel's current address in screen RAM. Since the Y register can only index up to 255 locations and we need to be able to access each of the 960 locations on the screen, the address of screen RAM on the zero page will be continually updated as the pinwheel is moved. We will still use indirect indexed addressing. But, instead of incrementing the Y register, we will add one to the screen RAM address of the pinwheel's current position.

Adding is done with the "ADC" instruction, which stands for ADd with Carry. ADC adds the value in the ADC instruction operand to the accumulator. ADC #$1, adds one to the value in the accumulator. The ADC instruction also includes the contents of the carry bit (in the status register) in the addition.

```
ADC #$1          $7C  Accumulator          $7C   Accumulator
                 $01  Add Operand          $01   Add Operand
                 $01  Carry Bit SET        $00   Carry Bit CLEAR
                 $7E                       $7D
```

Depending on whether the carry bit is set or not, the result of the addition will be $7E or $7D. The sum of the addition is always stored back into the accumulator. Unless you want to include the carry in an addition, you need to clear the carry bit to zero before adding. Clearing the carry bit will insure the accuracy of your addition. The "CLC" instruction is used to CLear the Carry flag of the status register. CLC uses implied addressing. No operand is needed. The assembly language code which adds one to the address of screen RAM is listed below.

```
LDA LOWSCR      ;LOAD THE ACC. WITH THE LOW BYTE OF SCREEN RAM
CLC             ;CLEAR THE CARRY BIT TO 0
ADC #$1         ;ADD 1 TO THE ACCUMULATOR
STA LOWSCR      ;STORE THE ACC. IN THE LOW BYTE OF SCREEN RAM
LDA HISCR       ;LOAD ACC. WITH THE HIGH BYTE OF SCREEN RAM
ADC #$00        ;ADD ZERO TO THE ACCUMULATOR
STA HISCR       ;STORE THE SUM IN HISCR
RTS             ;RETURN
```

Does it seem strange that one is added to LOWSCR and then zero is added to HISCR? Imagine the situation where LOWSCR is $FF and HISCR is $9C ($9CFF). Now add one to LOWSCR.

```
Carry Bit = 1          $ FF
                       $ 01
                       ----
                       $ 00
```

The answer stored in the accumulator will be zero and the carry bit is set. The new screen RAM address is $9C00. The high byte of the address,(9C), remains the same. However, $9C00 does not follow $9CFF in screen RAM -- $9D00 does. The carry bit needs to be added to the high order byte of the screen address. That explains the addition with HISCR. The carry bit was cleared before adding one to the low order byte of the address. If the carry was set by the first addition, a one will be included in the addition when zero is added to the high order byte of the address.

```
    LDA LOWSCR: ADC #$1       LDA HISCR: ADC #$00

Carry Bit=1 $ FF   LOWSCR       $9C HISCR
            $  1   ADC #$1      $00   ADC #$00
            $ 00   CLC            1   Carry
            ----               ----
            $ 00                $9D
                  \            /
                     $9D00
```

If the carry bit is not set by the first addition, zero is added to the high byte of the address, so it goes unchanged.

```
    LDA LOWSCR: ADC #$1       LDA HISCR: ADC #$00

Carry Bit=0 $ 40   LOWSCR       $9C HISCR
            $  1   ADC #$1      $00   ADC #$00
            $ 00   CLC            0   Carry
            ----               ----
            $ 41                $9C
                  \            /
                     $9C41
```

Turn to Assembly Language Programming Worksheet #14 to see how this addition routine can be incorporated into the program to make the pinwheel move to the right across the screen.

Assembly Language Programming Worksheet #14.

1. ENTER the ANIRIGHT program on your advanced topics diskette.

As your programs get longer and more complex, it becomes necessary to set up a "main loop," which "calls" each of the subroutines.

2. To see the main loop in the ANIRIGHT program, list lines 120-180.

Type: LIST 120,180 and press RETURN

You will notice a list of JSR's to different subroutines in the program. The main loop listed below has been inserted into the beginning of the program, following the constant and variable declarations.

```
BEGIN JSR DRAW       ;JUMP TO THE PINWHEEL DRAW
      JSR DELAY      ;PAUSE WHILE DISPLAY PINWHEEL
      JSR RIGHT      ;MOVE THE PINWHEEL TO THE RIGHT
      JMP BEGIN      ;JUMP BACK TO BEGIN AND
                     ;RE-EXECUTE THE LOOP
```

The first JSR DRAW draws the pinwheel in its starting position. The JSR DELAY holds the pinwheel in place momentarily, so we can see it before it is moved to the right. JSR RIGHT calls the routine that adds one to the address of the pinwheel's position on the screen. In order to see the pinwheel move, we want to draw the pinwheel again in its updated position. Instead of adding another JSR DRAW, the next instruction, JMP BEGIN, sends the CPU back to the label BEGIN, and the first JSR DRAW is re-executed. The screen address has been updated, so the pinwheel is drawn in its new location.

3. LIST 450-550 and you will see that the add routine has been incorporated into the program.

Type: LIST 450,550 and press RETURN

4. Don't forget that by adding the indirect indexed instruction, we have added another use of the Y register to the program. However, both the DRAW routine and the DELAY routine reset the Y register to zero. Thus, the additional use of the Y register does not effect the subroutines.

5. Assemble and execute the program from the debugger.

The main loop in this program is an infinite loop. To stop the program you need to press SYSTEM RESET. If you let the ANIRIGHT program continue past the last location in screen memory, the program will continue to store the code for the pinwheel in successive memory locations. The last address of the screen RAM is $9FFF. The assembler editor is stored in memory starting at $A000. If you let the ANIRIGHT program continue, you may write over the assembler editor in memory with pinwheel data. If this occurs, the EDIT prompt will not come on the screen when you press SYSTEM RESET. In that case, you will have to reboot the system.

6. Why are all those extra lines left on the screen?

Animating shapes in BASIC and assembly language requires the same sequence of steps.

1. Set up the location for the pinwheel on the screen.
2. Draw the shape.
3. Hold the shape on the screen with a delay.
4. Erase the shape.
5. Repeat the cycle.

The cycle is continued as long as the shape is being animated.

In the ANIRIGHT program, we need an erase routine to draw over the last line of the pinwheel, before a pinwheel is drawn in the next position on the screen. To erase the line we will store a space in the pinwheel's most recent position. Look over the ERASE routine listed below.

```
ERASE LDY #$00       ;INDEX FOR ZERO PAGE ADDRESSING
      LDA #$00       ;CHARACTER CODE FOR SPACE
      STA (LOWSCR),Y ;STORE OVER LAST PINWHEEL
      RTS            ;RETURN
```

The ERASE routine is really quite simple. Indexed indirect addressing is used to store the space in the pinwheel's most recent position. Turn to Assembly Language Worksheet #15 to see how the ANIRIGHT program has been changed by incorporating the ERASE routine.

Assembly Language Programming Worksheet #15

1. ENTER the program called ERASE on the advanced topics diskette.

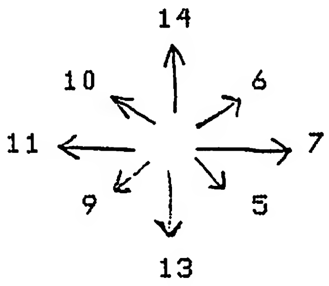
2. LIST lines 550-650 to see that the ERASE routine has been added. ERASE is called from the main loop.

   Type: LIST 550,650 and press RETURN

3. Assemble the program and run it from the debugger. Remember to press SYSTEM RESET to get back to the EDIT prompt. Otherwise, you will have to reboot the system.

4. When the pinwheel reaches the right edge of the screen, it comes back on the left side of the screen, one line down. What do you think causes the pinwheel to "wrap around" the screen?

Now let's add joystick control. To move the pinwheel with the joystick, you must first know which direction the joystick is being pushed. Values are assigned to the different positions of the joystick.



When the joystick is pushed to the right, the number 7 is stored in a memory location reserved for joystick feedback. Which memory location the 7 is stored in depends on which "port" (on the front of the Atari) the joystick is plugged into. If the joystick is plugged into the first port on the far left, the 7 will be stored in memory location $278 (632 in decimal). So to see which direction joystick #1 has been pushed, you simply need to read the contents of $278. The memory addresses reserved for feedback from the joysticks plugged into ports one through four are listed below.

Joystick in Port #1	$278
Joystick in Port #2	$279
Joystick in Port #3	$27A
Joystick in Port #4	$27B

One way to read the contents of a memory location is to load the accumulator with the value and do a series of comparisons. For example, LDA $278 loads the accumulator with the most recently depressed direction of joystick #1. To check the value we can compare the accumulator with the specific values we are looking for. If we compare the contents of the accumulator with 7 and find that the value is 7, we know that the joystick has been pressed to the right. An assembly language routine that compares the joystick reading with the values for left and right is listed below.

```
LDA #$278       ;READ JOYSTICK PORT #1
CMP #$7         ;IS IT A 7?
BEQ RIGHT       ;IF SO, BRANCH TO THE RIGHT ROUTINE
CMP #$B         ;IS IT 11
BEQ LEFT        ;IF SO, BRANCH TO THE LEFT ROUTINE
```

Comparisons are only made with those values for the directions we are looking for. Any other value returned from the joystick in $278 is ignored. Thus, if the joystick is pressed on a diagonal, a 6 will be loaded into the accumulator. When the comparisons are made for a left or a right joystick press, the 6 will be ignored since the 6 does not match the 7 for right, or the 11 for left.

RIGHT and LEFT are labels for subroutines which change the pinwheel's direction of travel. Turn to Assembly Language Worksheet #15 to see how the joystick reading routine can be incorporated into the program.

Assembly Language Programming Worksheet #15

1. ENTER the JOYMOVE program on your advanced utilities diskette.

2. LIST lines 150-220.

Type: LIST 150,220 and press RETURN

A JSR JOYSTICK command has been added to the main loop, and the RIGHT routine is no longer there. The JOYSTICK routine gets directional feedback from the joystick. Instead of being called from the main loop, the RIGHT routine is called from the JOYSTICK routine, whenever the person using the program pushes the joystick to the right.

3. LIST lines 100-150 to see how the name "STICK" has been assigned to the address $278 in the constant and variable declarations at the top of the program. For anyone reading through the program the name STICK is much easier to understand than the hexadecimal address $278.

4. We have a routine to move the pinwheel to the right. Now we need a routine to move the pinwheel to th left. Since the addresses of each row of screen memory are numbered from left to right, instead of adding, we need to subtract one from the screen address in order to move the pinwheel to the left.

```
         -------------------------
$9C40 | Subtract <- * -> Add    |$9C68
      |             ^            |
      |       You are here       |
      |                          |
         -------------------------
```

When we wrote the add routine, first we had to clear the carry bit of the status register with the CLC instruction. The opposite is true for subtraction. Before subtracting you need to set the carry bit with an "SEC" instruction. This is due to a peculiarity of the CPU's numbering system. If you would like an explanation of why you must set the carry bit before subtracting, see Chapter 9 of The Atari Assembler, by Don and Kurt Inman. There are copies in the camp library.

The format of the subtraction subroutine is identical to the addition routine. The carry bit is set with SEC. The "SBC", SuBtract with Carry instruction, subtracts the number in its operand from the accumulator. The result is stored back in the accumulator. "Double precision" arithmetic, where the high byte of an address must be updated based on the results of the low byte arithmetic, is repeated in this routine. Try writing your own routine which moves the pinwheel to the left.

5. LIST lines 300-380 to review the RIGHT routine. Now try writing a left routine below.

```
LEFT _______________;LOAD THE ACC. WITH LOWSCR

     _____SEC_______;SET THE CARRY BIT

     _______________;SUBTRACT $1 FROM THE ACCUMLATOR

     _______________;STORE THE ANSWER IN LOWSCR

     _______________;LOAD THE ACCUMULATOR WITH HISCR

     _______________;SUBTRACT ZERO FROM VALUE IN ACC.

     _______________;STORE THE ANSWER IN HISCR

     _______________;RETURN
```

LIST lines 390-460, to compare your subroutine with the LEFT routine in the JOYMOVE program.

6. Assemble the program and run it from the debugger. You should be able to move the pinwheel to the right or left with the joystick. Since there is no UP or DOWN routine, the pinwheel will not respond when you press the joystick in those directions. The program is in a continuous loop, which reads the joystick and moves the pinwheel continuously. You must press SYSTEM RESET to stop the program. You will be returned to the editor. How can you change the program so that it is not an infinite loop?

----

7. LIST lines 90-150. Note that a JMP BEGIN command has been added. The assembler goes through two steps to assemble an assembly language program. First, it reads through the program and assigns memory addresses to each of the constants, variables, and labels. In this first step a "symbol table" of the addresses is compiled by the assembler. Some assemblers list the symbol table after a program is assembled. If not, the symbol table remains hidden from the assembler user, as is the case with the Atari assembler. The assemblers second step is to execute each instruction in the program starting with the first instruction in the object code. The JMP instruction tells the assembler to jump over the constant and variable declarations at the beginning of the program and go directly to the first instruction of the program. This is not a essential procedure and it will not affect the performance of your program. Some programmers like to insert the JMP instruction in the beginning of their programs for style and clarity.

Assembly Language Programming Worksheet #16

Now all we need are two routines that move the pinwheel up and down.

1. The subroutine that moves the pinwheel down one line is identical to the RIGHT routine, except for the number that is added to the LOWSCR address. If there are forty spaces per line, how much should be added to the LOWSCR address to move the pinwheel down one row?_______

2. LIST lines 300-380 of the JOYMOVE program to review the RIGHT routine. Try writing your own DOWN routine. Fill in the blanks below.

```
     DOWN   ______________;LOAD THE ACCUMULATOR WITH LOWSCR

            ______________;CLEAR THE CARRY

            ______________;ADD ONE TO ACC.,INCLUDE THE CARRY

            __STA LOWSCR__; _________________________

            ______________;LOAD THE ACCUMULATOR WITH HISCR

            ___ADC #$00___; _________________________

            ______________;STORE THE ACCUMULATOR IN HISCR

            _____________;RETURN
```

3. Now write a routine that will move the pinwheel UP the screen.

```
      UP    ______________;LOAD THE ACCUMULATOR WITH LOWSCR

            _____SEC______;SET THE CARRY BIT

            ______________;SUBTRACT ONE TO THE ACCUMULATOR

            ______________;STORE THE ACCUMULATOR IN LOWSCR

            ______________;LOAD THE ACCUMULATOR WITH HISCR

            ______________;ADD ZERO AND THE CARRY BIT TO HISCR

            ______________;STORE THE ACCUMULATOR IN HISCR

            ______________;RETURN
```

4. The last set of instructions that need to be updated before the animation is complete is the joystick routine. The UP and DOWN routines need to be included in the JOYSTICK reading routine. The current listing of the JOYSTICK routine is printed below. Complete the comparisons and branches to the UP and DOWN routines.

```
    JOYSTICK LDA STICK    ;LOAD ACC. WITH JOYSTICK PRESS
             CMP #$7      ;COMPARE THE FEEDBACK TO 7 - RIGHT
             BEQ RIGHT    ;IF EQUAL TO 7 THEN BRANCH TO RIGHT
             CMP #$B      ;TO THE LEFT?
             BEQ LEFT     ;IF SO BRANCH TO LEFT ROUTINE

             ______ #$D   ;IS THE FEEDBACK EQUAL TO 14

             ______ UP    ;IF SO, THEN BRANCH TO UP

             ______ #$C   ;IS THE ACCUMULATOR = 13?

             ______ DOWN ;IF SO THEN BRANCH TO DOWN

             ___________ ;RETURN
```

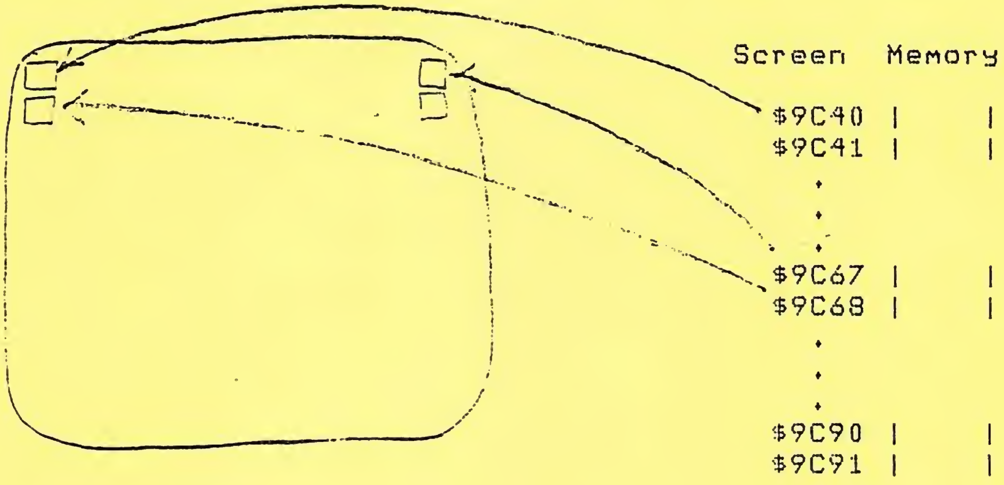
5. ENTER the ANIMATE program on your advanced topics diskette.

6. LIST lines 510-660 to compare your DOWN and UP routines with the ones in the ANIMATE program. You will not be able to see the entire listing at once. Instead you will have to list the subroutines individually.

7. LIST 250-350 to check your JOYSTICK routine against the one in the ANIMATE program.

8. And finally - assemble the program and try it out.

The pinwheel moves in each of the four directions. When you move it left or right and the pinwheel goes off the screen, it comes back on the screen on the opposite side. This is because screen memory is sequential from one row to the next on the screen. The address of the rightmost position on the top row of the screen is just before the leftmost position on the second row of the screen.



When you move the joystick up or down off the screen, peculiar things happen on the screen. This is because the pinwheel has moved out of screen RAM and is storing pinwheel data in areas of memory being used for other purposes. The program never checks where the pinwheel is in memory, it just adds or subtracts 40 from the pinwheel's current address or position. Remember, all that exists in memory is a long string of boxes, each holding one number. It is the sequence ad the CPU's interpretation of those numbers, that enables the computer to do such amazing things. If we store the values for the pinwheel and then erase the pinwheel in memory locations outside of screen RAM we are leaving zeros in areas of memory that might have held important data or instructions for the CPU. Thus, when you move the pinwheel up or down off the screen, you may be writing over the data in memory, which is there for other purposes, and you may confuse the computer so much that SYSTEM RESET will not return you to the EDIT prompt. Instead, you will have to reboot the system.

In conclusion, we have set aside areas of memory to serve different functions. The zero page holds the screen RAM address, which we access with indirect indexed addressing. Memory locations $600-$689 hold our program. And we are using memory locations $9C40-$9FFF to hold the data for what is being displayed on the screen. While the numbers in these memory locations bear significance to us, the programmers and the CPU, to someone who is unfamiliar with computers or assembly language programming memory contains just a long, LONG, list of unintelligible numbers.

```
Use of Memory                                        Contents of Memory

$0000  |      |                                      $0000 | 00 |
  .                                                    .
  .                                                    .
  .                                                    .
$0058  |LOWSCR|  } Zero Page                         $0058 | 40 |
$0059  |HISCR |                                      $0059 | 9C |
  .                                                    .
  .                                                    .
  .                                                    .
$0600  | JMP  |                                      $0600 | 4C |
$0601  |BEGIN |                                      $0601 | 03 |
$0602  |ADDRES|                                      $0602 | 06 |
  .                                                    .
  .                                                    .
  .                                                    .
$0650  | LDA  |                                      $0650 | A5 |
$0651  |LOWSCR|  } Animate                           $0651 | 58 |
$0652  | CLC  |    Program                           $0652 | 18 |
  .                                                    .
  .                                                    .
  .                                                    .
$0686  | DEX  |                                      $0686 | CA |
$0687  | BNE  |                                      $0687 | D0 |
$0688  |OFFSET|                                      $0688 | F8 |
  .                                                    .
  .                                                    .
  .                                                    .
$9C40  |  7C  |                                      $9C40 | 7C |
$9C41  |      |                                      $9C41 | 00 |
$9C42  |      |                                      $9C42 | 00 |
  .                                                    .
  .              } Screen Ram                          .
  .                                                    .
$9CFE  |      |                                      $9CFE | 00 |
$9CFF  |      |                                      $9CFF | 00 |
```

Summary and Challenges

6502 Addressing Modes

```
Immediate:     LDA #$50   ;Load the accumulator with
                           immediate $50.

Absolute:      LDA $278   ;Load the accumulator with the
                           contents of memory location
                           $278.

Zero Page:     LDA $80    ;Load the accumulator with the
                           contents of the zero page
                           location $80.

  Zero Page,X:     LDA $58,X    ;Load the accumulator with
                                 the contents of $58+X.

  Zero Page,Y:     LDA $58,Y    ;Load the accumulator
                                 with the contents of $58+Y.

Implied:       CLC        ;Clear the carry bit.  Increment
                           the X register by one.

Relative:      BNE WAIT   ;Branch to WAIT as long as the zero
                           bit of the status register is not
                           set.  Branches are made relative to
                           the instructions being branched to.
                           The CPU will not let you branch
                           further than 127 bytes.  Branch
                           instructions are the only
instructions that use
                           relative addressing.

Indexed:       LDA $9C40,X ;Add the value in X to $9C40.
                             Load the accumulator with the

                             contents of the total
                             of ($9C40+X).

               LDA SCREEN,Y ;Add the contents of the Y
                              register to the address assigned
                              to the label SCREEN.
                              Load the accumulator with the
                              contents of the new address.
```

```
Indirect Indexed:   LDA ($58),Y ;Get the address stored in
                                 $58 and $59 on the zero page
                                   of memory. Add the value
                                   in Y to the address.
                                   Load the accumulator
                                   with the contents
                                   of the new address.

Indexed Indirect:   LDA ($58,X) ;Add the value in X to $58.
                                   Suppose X is 2,
                                   X + $58 = $5A.
                                   Get the address
                                   stored on the zero page
                                   in $5A and $5B.
                                   Load the accumulator
                                   with the contents of
                                   the address stored
                                   in $60 and $61.
```

The appendices of the Atari Assembler Editor Manual include listings of the 6502 instruction set and their corresponding addressing modes and opcodes.

The appendices of The Atari Assembler, by Don and Kurt Inman, include the 6502 instruction set, addressing modes, opcodes, and the status flags affected by each instruction. You can find a copy of The Atari Assembler in the camp library.

To learn how to save your assembly language programs on disk, see pages 19-23 of the Atari Assembler Editor User's Manual.

Challenges

1. Write an assembly language program that prints your name in the middle of the screen. Use the .BYTE psuedo opcode and indexed addressing to print your name.

2. In the animation programs, we are continually changing the address of screen RAM held in $58 and $59 to the updated position of the pinwheel. Memory locations $58 and $59 are the locations the computer uses to hold the starting address of screen RAM. When a break occurs in the animation program, the computer uses the address it finds in $58 and $59 for the starting location on the screen. Consequently, after a break in an animation program, the screen looks as though it has new margins and print is oddly formated on the screen because the address in $58 an $59 was the last position of the pinwheel. Edit the ANIMATE program so that the address in $58 and $59 will be preserved. Store the starting address of screen RAM in two consecutive memory locations on the zero page. Memory locations $CB-$CF are free bytes of memory. Whenever the pinwheel is moved, update your own screen address rather than interfering with the address stored at $58 and $59.

3. Instead of leaving a zero in each of the pinwheel's last locations in order to erase the last line of the pinwheel, save what was stored in the screen memory location before putting the pinwheel there. Save the original contents of the memory location on the stack, draw the pinwheel, and then recover the original contents of memory to its former location. For example, if there is an A displayed on the screen and the pinwheel is about to move into the A's position, push the A onto the stack, and then display the pinwheel. Then pull the A off the stack and store it back in its original screen memory location. This way the pinwheel will not erase everything in its path. Instead the screen display will be left intact.

4. Add some comparisons to the direction subroutines that stop the pinwheel at the edge of the screen. Do not let it wrap around or write over memory above or below screen RAM.

5. Read the joystick for diagonal joystick presses. Incorporate the necessary routines to move the pinwheel on a diagonal as well as up and down and left and right.

6. Animate a shape made up of keyboard control characters, which is three or four characters wide and high.

```
              25 ;          ARROW
              50 ;
0000          0100            *=     $0600       ;ORIGIN OF PROGRAM
0600 A97D     0110            LDA    #$7D        ;LOAD ACC. WITH ARROW
0602 8D409C   0120            STA    $9C40       ;SCREEN RAM LOCATION
0605 60       0130            RTS                ;RETURN FROM SUBROUTINE
```

```
              25 ;            ARW2
              50 ;
0000          0100            *=     $0600       ;ORIGIN OF PROGRAM
0600 A97D     0110            LDA    #$7D        ;LOAD ACC. WITH ARROW
0602 8D409C   0120            STA    $9C40       ;SCREEN RAM LOCATION
0605 A900     0130            LDA    #$00        ;LOAD ACC. WITH SPACE
0607 8D409C   0140            STA    $9C40       ;STORE SPACE OVER ARROW
060A 60       0150            RTS                ;RETURN FROM SUBROUTINE
```

```
              25 ;            SCRADR
              50 ;
0000          0100            *=     $0600       ;ORIGIN OF PROGRAM
9C40          0105 SCREEN     =      $9C40       ;ASSIGN SCREEN
0600 A97D     0110            LDA    #$7D        ;LOAD ACC. WITH ARROW
0602 8D409C   0120            STA    SCREEN      ;STORE ACC. ON SCREEN
0605 60       0130            RTS                ;RETURN FROM SUBROUTINE
```

```
              25 ;            HOLDARROW
              50 ;
0000          0100            *=     $600            ;ORIGIN
9C40          0110 SCREEN     =      $9C40
0600 A000     0120            LDY    #$00         ;SET COUNTER
0602 A97D     0130            LDA    #$7D         ;CODE FOR ARROW
0604 8D409C   0140            STA    SCREEN       ;DISPLAY
0607 C8       0150 DELAY      INY                ;ADD ONE TO Y,COUNTER
0608 D0FD     0160            BNE    DELAY         ;IF NOT 0, THEN REPEAT DELAY
060A 60       0170            RTS                 ;RETURN
```

```
              10 ;           PINWHEEL
              15 ;
              20 ;THIS PROGRAM USES THE .BYTE
              25 ;PSUEDO OPCODE TO STORE DATA
              30 ;IN MEMORY AND INDEXED ADDRESSING
              35 ;TO READ THROUGH THE DATA.
              40 ;THE PURPOSE OF THE PROGRAM IS
              45 ;TO DISPLAY A SPINNING PINWHEEL
              50 ;IN THE UPPER LEFT HAND CORNER
              55 ;OF THE SCREEN.
              60 ;
              65 ;
0000          0100          *=   $600           ;ORIGIN
9C40          0110 SCREEN =      $9C40     ;SCREEN RAM
0600 A200     0120          LDX  #$00          ;SET INDEX TO 0
0602 BD0E06   0130 NEXTCHAR LDA CHAR,X     ;GET NEXT CHAR
0605 8D409C   0140          STA  SCREEN        ;DISPLAY IT
0608 E8       0150          INX               ;ADD ONE TO INDEX
0609 E004     0160          CPX  #$4           ;COMPARE X REG. TO 4
060B D0F5     0170          BNE  NEXTCHAR      ;IF X=4 THEN BRANCH FOR CHAR
060D 60       0180          RTS               ;RETURN
060E 7C       0190 CHAR     .BYTE 124,15,13,60  ;PINWHEEL
060F 0F
0610 0D
0611 3C
```

```
              10 ;            SUBROUTINE
              20 ;THIS PROGRAM PRINTS AN ARROW IN
              30 ;THE UPPER LEFT HAND CORNER OF THE
              40 ;SCREEN.  A CALL TO A DELAY LOOP
              50 ;HOLDS THE ARROW ON THE SCREEN
              60 ;
              70 ;
0000          0100          *=   $600
9C40          0110 SCREEN =      $9C40
0600 A97D     0120          LDA  #$7D       ;CODE FOR AN ARROW
0602 8D409C   0130          STA  SCREEN     ;DISPLAY
0605 200906   0140          JSR  DELAY      ;WAIT
0608 60       0150          RTS             ;RETURN
              0160 ;
              0170 ;
              0180 ;
0609 A2A0     0190 DELAY    LDX  #$A0       ;COUNTER FOR Y LOOPS
060B A000     0200 AGAIN    LDY  #$00       ;0-FF COUNTER
060D C8       0210 WAIT     INY             ;ADD ONE TO Y
060E D0FD     0220          BNE  WAIT        ;IF NOT 0, REPEAT WAIT
0610 CA       0230          DEX             ;SUBTRACT ONE FROM X
0611 D0F8     0240          BNE  AGAIN       ;IF NOT 0, REPEAT AGAIN
0613 60       0250          RTS             ;RETURN
```

```
               10 ;          SPIN
               20 ;
               30 ;THIS PROGRAM USES FOUR LINES
               40 ;TO PRINT A SPINNING PINWHEEL
               50 ;IN THE UPPER LEFT HAND CORNER
               60 ;OF THE SCREEN.  THE PINWHEEL
               70 ;SPINS ONCE.
               80 ;
               90 ;
               0100 ;
               0110 ;
0000           0120        *=    $600              ;ORIGIN
9C40           0130 SCREEN =     $9C40        ;SCREEN RAM
0600 A200      0140 DRAW   LDX   #$00            ;SET INDEX TO 0
0602 BD1506    0150 NEXTCHAR LDA CHAR,X       ;GET NEXT CHAR
0605 8D409C    0160        STA   SCREEN          ;DISPLAY IT
0608 8A        0170        TXA                  ;TRANSFER X TO ACC.
0609 48        0180        PHA                  ;PUSH ACC. ONTO STACK
060A 201906    0190        JSR   DELAY           ;CALL DELAY LOOP
060D 68        0200        PLA                  ;PULL ACC. OFF STACK
060E AA        0210        TAX                  ;TRANSFER ACC. TO X
060F E8        0220        INX                  ;ADD ONE TO INDEX
0610 E004      0230        CPX   #$4             ;COMPARE X REG. TO 4
0612 D0EE      0240        BNE   NEXTCHAR        ;IF X=4 THEN BRANCH FOR CHAR
0614 60        0250        RTS                  ;RETURN
0615 7C        0260 CHAR   .BYTE 124,15,13,60   ;PINWHEEL
0616 0F
0617 0D
0618 3C
0619 A255      0270 DELAY  LDX   #$55         ;COUNT 0-255, $55 TIMES
061B A000      0280 AGAIN  LDY   #$00         ;SET COUNTER TO 0
061D C8        0290 WAIT   INY                ;INCREMENT Y REG.
061E D0FD      0300        BNE   WAIT              ;IF NOT 0,WAIT
0620 CA        0310        DEX                  ;SUBTRACT 1 FROM X
0621 D0F8      0320        BNE   AGAIN           ;IF NOT 0,AGAIN
0623 60        0330        RTS                  ;RETURN
```

```
            10 ;             ANIRIGHT
            20 ;
            30 ;  THIS PROGRAM MOVES THE SPINNING
            40 ;  PINWHEEL TO THE RIGHT, BY
            50 ;  CONTINUALLY INCREMENTING THE
            60 ;  SCREEN RAM POSITION.
            70 ;
            80 ;
0000        90        *=    $600
0058        0100 LOWSCR =     $58       ;LOW BYTE OF SCREEN RAM
0059        0110 HISCR  =     $59       ;HIGH BYTE OF SCREEN RAM
            0120 ;
            0130 ;  MAIN LOOP
            0140 ;
0600 200C06 0150 BEGIN  JSR  DRAW       ;DRAW THE PINWHEEL
0603 203406 0160        JSR  DELAY      ;HOLD ON THE SCREEN MOMENTARILY
0606 202606 0170        JSR  RIGHT      ;INCREMENT POSITION TO THE RIGHT
0609 4C0006 0180        JMP  BEGIN      ;REPEAT MAIN LOOP
            0190 ;
            0200 ;
            0210 ;  DRAW READS CHAR DATA AND
            0220 ;  PLACES LINES ON SCREEN IN
            0230 ;  SEQUENCE TO APPEAR LIKE
            0240 ;  SPINNING PINWHEEL.
            0250 ;
            0260 ;
060C A200   0270 DRAW   LDX  #$00       ;SET INDEX TO 0
060E A000   0280        LDY  #$00       ;SET INDEX TO 0
0610 BD2206 0290 NEXTCHAR LDA CHAR,X    ;INDEXED ADDRESSING, GET DATA
0613 9158   0300        STA  (LOWSCR),Y  ;INDIRECT INDEXED ADDRESSING TO SCREEN
0615 8A     0310        TXA             ;TRANSFER X REG. TO ACC.
0616 48     0320        PHA             ;PUSH ACC. ONTO STACK
0617 203406 0330        JSR  DELAY      ;CALL THE DELAY ROUTINE
061A 68     0340        PLA             ;PULL ACC OFF STACK
061B AA     0350        TAX             ;TRANSFER ACC. TO X REG.
061C E8     0360        INX             ;INCREMENT X REGISTER
061D E004   0370        CPX  #$4        ;4 LINES IN PINWHEEL
061F D0EF   0380        BNE  NEXTCHAR   ;GET NEXT CHAR
0621 60     0390        RTS             ;RETURN
0622 7C     0400 CHAR   .BYTE 124,15,13,60 ;PINWHEEL
0623 0F
0624 0D
0625 3C
            0410 ;
            0420 ; RIGHT ADDS ONE TO THE SCREEN
            0430 ; ADDRESS OF THE PINWHEEL
            0440 ;
            0450 ;
0626 A558   0460 RIGHT  LDA  LOWSCR      ;GET LOW BYTE OF SCREEN RAM
0628 18     0470        CLC                ;CLEAR THE CARRY
0629 6901   0480        ADC  #$1             ;ADD 1 AND CARRY TO ACC.
```

```
062B 8558    0490          STA  LOWSCR       ;UPDATE LOWSCR
062D A559    0500          LDA  HISCR        ;GET HIGH BYTE OF SCREEN RAM
062F 6900    0510          ADC  #$00         ;ADD 0 AND CARRY
0631 8559    0520          STA  HISCR        ;UPDATE HIGH BYTE SCREEN RAM
0633 60      0530          RTS              ;RETURN
             0540 ;
             0550 ;
             0560 ;  DELAY HOLDS THE IMAGE
             0570 ;  IN ONE PLACE, MOMENTARILY
             0580 ;  BEFORE THE NEXT MOVE.
             0590 ;
             0600 ;
0634 A219    0610 DELAY  LDX  #$19        ;COUNT 0-255, 25 TIMES
0636 A000    0620 AGAIN  LDY  #$00        ;SET COUNTER TO 0
0638 C8      0630 WAIT   INY              ;ADD 1 TO Y REG.
0639 D0FD    0640          BNE  WAIT           ;IF NOT 0, WAIT
063B CA      0650          DEX               ;SUBTRACT 1 FROM X REG.
063C D0F8    0660          BNE  AGAIN          ;$19 YET?
063E 60      0670          RTS               ;RETURN
```

```
            10 ;               ERASE
            20 ;
            30 ;  THIS PROGRAM MOVES THE SPINNING
            40 ;  PINWHEEL TO THE RIGHT, BY
            50 ;  CONTINUALLY INCREMENTING THE
            60 ;  SCREEN RAM POSITION.  EACH TIME
            70 ;  THE PINWHEEL IS DRAWN, A SPACE
            80 ;  IS PRINTED OVER THE LAST PINWHEEL
            90 ;  POSITION SO NOT TO LEAVE A TRAIL
            0100 ;
            0110 ;
0000        0120        *=   $600
0058        0130 LOWSCR =    $58       ;LOW BYTE OF SCREEN
0059        0140 HISCR  =    $59       ;HIGH BYTE OF SCREEN RAM
            0150 ;
            0160 ;  MAIN LOOP
            0170 ;
0600 200F06 0180 BEGIN  JSR  DRAW      ;DRAW THE PINWHEEL
0603 203E06 0190        JSR  DELAY     ;HOLD ON THE SCREEN MOMENTARILY
0606 203706 0200        JSR  ERASE     ;ERASE LINE WITH SPACE
0609 202906 0210        JSR  RIGHT     ;INCREMENT POSITION TO THE RIGHT
060C 4C0006 0220        JMP  BEGIN     ;REPEAT MAIN LOOP
            0230 ;
            0240 ;
            0250 ;  DRAW READS CHAR DATA AND
            0260 ;  PLACES LINES ON SCREEN IN
            0270 ;  SEQUENCE TO APPEAR LIKE
            0280 ;  SPINNING PINWHEEL.
            0290 ;
            0300 ;
060F A200   0310 DRAW   LDX  #$00      ;SET INDEX TO 0
0611 A000   0320        LDY  #$00      ;SET INDEX TO 0
0613 BD2506 0330 NEXTCHAR LDA CHAR,X   ;INDEXED ADDRESSING, GET DATA
0616 9158   0340        STA  (LOWSCR),Y  ;INDIRECT INDEXED ADDRESSING TO SCREEN
0618 8A     0350        TXA            ;TRANSFER X REG. TO ACC.
0619 48     0360        PHA            ;PUSH ACC. ONTO STACK
061A 203E06 0370        JSR  DELAY     ;CALL THE DELAY ROUTINE
061D 68     0380        PLA            ;PULL ACC OFF STACK
061E AA     0390        TAX            ;TRANSFER ACC. TO X REG.
061F E8     0400        INX            ;INCREMENT X REGISTER
0620 E004   0410        CPX  #$4       ;4 LINES IN PINWHEEL
0622 D0EF   0420        BNE  NEXTCHAR  ;GET NEXT CHAR
0624 60     0430        RTS            ;RETURN
0625 7C     0440 CHAR   .BYTE 124,15,13,60 ;PINWHEEL
0626 0F
0627 0D
0628 3C
            0450 ;
            0460 ; RIGHT ADDS ONE TO THE SCREEN
            0470 ; ADDRESS OF THE PINWHEEL
            0480 ;
            0490 ;
0629 A558   0500 RIGHT  LDA  LOWSCR    ;GET LOW BYTE OF SCREEN RAM
062B 18     0510        CLC              ;CLEAR THE CARRY
062C 6901   0520        ADC  #$1          ;ADD 1 AND CARRY TO ACC.
062E 8558   0530        STA  LOWSCR       ;UPDATE LOWSCR
0630 A559   0540        LDA  HISCR        ;GET HIGH BYTE OF SCREEN RAM
0632 6900   0550        ADC  #$00         ;ADD 0 AND CARRY
0634 8559   0560        STA  HISCR        ;UPDATE HIGH BYTE SCREEN RAM
0636 60     0570        RTS              ;RETURN
            0580 ;
            0590 ;
            0600 ;  ERASE PUTS A SPACE OVER THE
            0610 ;  SPINNING PINWHEEL'S LAST POSITION.
            0620 ;
            0630 ;
0637 A000   0640 ERASE  LDY  #$00      ;INDEX
0639 A900   0650        LDA  #$00         ;VALUE FOR SPACE
063B 9158   0660        STA  (LOWSCR),Y   ;STORE IN LAST LOCATION
063D 60     0670        RTS              ;RETURN
            0680 ;
            0690 ;
            0700 ;  DELAY HOLDS THE IMAGE
            0710 ;  IN ONE PLACE, MOMENTARILY
            0720 ;  BEFORE THE NEXT MOVE.
            0730 ;
            0740 ;
063E A225   0750 DELAY  LDX  #$25      ;COUNT 0-255, $25 TIMES
0640 A000   0760 AGAIN  LDY  #$00      ;SET COUNTER TO 0
0642 C8     0770 WAIT   INY            ;ADD 1 TO Y REG.
0643 D0FD   0780        BNE  WAIT         ;IF NOT 0, WAIT
0645 CA     0790        DEX              ;SUBTRACT 1 FROM X REG.
0646 D0F8   0800        BNE  AGAIN        ;$19 YET?
0648 60     0810        RTS              ;RETURN
```

```
             10 ;                   JOYMOVE
             20 ;
             30 ;  THIS PROGRAM MOVES A SPINNING PINWHEEL TO THE LEFT
             40 ;  OR THE RIGHT ON THE SCREEN.  THE PINWHEEL'S
             50 ;  DIRECTION OF TRAVEL IS CONTROLLED
             60 ;  BY THE JOYSTICK IN PORT #1.
             70 ;
             80 ;
0000         90             *=   $600
0600 4C0306 0100            JMP  BEGIN      ;JUMP OVER VARIABLES AND CONSTANTS
0278        0110 STICK  =        $278       ;FEEDBACK FROM JOYSTICK #1
0058        0120 LOWSCR =        $58        ;LOW BYTE OF SCREEN RAM
0059        0130 HISCR  =        $59        ;HIGH BYTE OF SCREEN RAM
            0140 ;
            0150 ; MAIN LOOP
            0160 ;
0603 201206 0170 BEGIN  JSR  JOYSTICK       ;READ JOYSTICK SUBROUTINE
0606 203A06 0180        JSR  DRAW           ;DRAW THE PINWHEEL
0609 205B06 0190        JSR  DELAY          ;LEAVE ON THE SCREEN MOMENTARILY
060C 205406 0200        JSR  ERASE          ;ERASE WITH A SPACE
060F 4C0306 0210        JMP  BEGIN          ;JUMP TO BEGIN, REPEAT MAIN LOOP
            0220 ;
            0230 ;  READ AND INTERPRET THE VALUE RETURNED FROM THE JOYSTICK
            0240 ;
0612 AD7802 0250 JOYSTICK LDA STICK       ;LOAD ACC WITH CONTENTS OF $278
0615 C907   0260        CMP  #$7            ;WAS IT PRESSED TO THE RIGHT?
0617 F005   0270        BEQ  RIGHT          ;IF YES BRANCH TO RIGHT ROUTINE
0619 C90B   0280        CMP  #$B            ;TO THE LEFT?
061B F00F   0290        BEQ  LEFT           ;IF SO BRANCH TO LEFT ROUTINE
061D 60     0300        RTS
061E A558   0310 RIGHT  LDA  LOWSCR         ;GET LOW BYTE OF SCREEN RAM
0620 18     0320        CLC                 ;CLEAR THE CARRY BIT
0621 6901   0330        ADC  #$1            ;ADD 1 AND CARRY TO ACC.
0623 8558   0340        STA  LOWSCR         ;UPDATE LOWSCR
0625 A559   0350        LDA  HISCR          ;GET HIGH BYTE
0627 6900   0360        ADC  #$00           ;ADD CARRY AND ZERO TO HIGH BYTE
0629 8559   0370        STA  HISCR
062B 60     0380        RTS
062C A558   0390 LEFT   LDA  LOWSCR         ;GET LOW BYTE OF SCREEN RAM
062E 38     0400        SEC                 ;SET THE CARRY BIT
062F E901   0410        SBC  #$1            ;SUBTRACT 1 AND CARRY
0631 8558   0420        STA  LOWSCR
0633 A559   0430        LDA  HISCR          ;GET HIGH BYTE SCREEN RAM
0635 E900   0440        SBC  #$00           ;ANYTHING IN CARRY TO SUBTRACT?
0637 8559   0450        STA  HISCR          ;UPDATE HIGH BYTE SCREEN RAM
0639 60     0460        RTS
            0470 ;
            0480 ;  DRAW READS CHAR DATA AND PLACES LINES
            0490 ;  ON SCREEN IN ORDER OF SEQUENCE TO APPEAR LIKE
            0500 ;  A SPINNING PINWHEEL
            0510 ;
```

```
063A A200   0520 DRAW    LDX  #$00           ;SET INDEX TO 0
063C A000   0530         LDY  #$00           ;INDEX
063E BD5006 0540 NEXTCHR LDA CHAR,X          ;INDEXED ADDRESSING
0641 9158   0550         STA  (LOWSCR),Y     ;INDEXED INDIRECT ADDRESSING
0643 8A     0560         TXA                 ;TRANSFER X TO ACC.
0644 48     0570         PHA                 ;PUSH ACC. ONTO STACK
0645 205B06 0580         JSR  DELAY          ;JUMP TO DELAY ROUTINE
0648 68     0590         PLA                 ;PULL ACC. OFF STACK
0649 AA     0600         TAX                 ;TRANSFER ACC. TO X REG.
064A E8     0610         INX                 ;INCREMENT X
064B E004   0620         CPX  #$4            ;4 LINES IN PINWHEEL
064D D0EF   0630         BNE  NEXTCHR        ;GET NEXT ONE
064F 60     0640         RTS
0650 7C     0650 CHAR    .BYTE 124,15,13,60 ;VALUES FOR LINES
0651 0F
0652 0D
0653 3C
            0660 ;
            0670 ;  ERASE PUTS A SPACE OVER THE SPINNING
            0680 ;  PINWHEELS LAST POSITION
            0690 ;
0654 A000   0700 ERASE   LDY  #$00           ;INDEX FOR ZERO PAGE ADDRESSING
0656 A900   0710         LDA  #$00           ;VALUE FOR SPACE
0658 9158   0720         STA  (LOWSCR),Y     ;STORE IN LAST LOCATION
065A 60     0730         RTS
            0740 ;
            0750 ;  DELAY HOLDS THE IMAGE IN ONE PLACE MOMENTARILY
            0760 ;  BEFORE READING NEXT MOVE
            0770 ;
065B A219   0780 DELAY   LDX  #$19        ;COUNT 0-255 25 TIMES
065D A000   0790 AGAIN   LDY  #$00
065F C8     0800 WAIT    INY             ;INCREMENT Y REGISTER
0660 D0FD   0810         BNE  WAIT           ;IF NOT ZERO, WAIT
0662 CA     0820         DEX                 ;25 YET?
0663 D0F8   0830         BNE  AGAIN          ;IF NOT ZERO, AGAIN
0665 60     0840         RTS
```

```
            10 ;                 ANIMATE
            20 ;
            30 ;THIS PROGRAM MOVES A SPINNING PINWHEEL AROUND ON THE
            40 ;GRAPHICS ZERO SCREEN.  THE PINWHEEL IS CONTROLLED BY A
            50 ;JOYSTICK PLUGGED INTO PORT #1
            60 ;
            70 ;
            80 ;
            90 ;
0000        0100        *=   $600
0600 4C0306 0110        JMP  BEGIN     ;JUMP OVER VARIABLES AND CONSTANTS
0278        0120 STICK  =    $278      ;FEEDBACK FROM JOYSTICK #1
0058        0130 LOWSCR =    $58       ;LOW BYTE OF SCREEN RAM
0059        0140 HISCR  =    $59       ;HIGH BYTE OF SCREEN RAM
            0150 ;
            0160 ; MAIN LOOP
            0170 ;
0603 201206 0180 BEGIN  JSR  JOYSTICK      ;READ JOYSTICK SUBROUTINE
0606 205E06 0190        JSR  DRAW          ;DRAW THE PINWHEEL
0609 207F06 0200        JSR  DELAY         ;LEAVE ON THE SCREEN MOMENTARILY
060C 207806 0210        JSR  ERASE         ;ERASE WITH A SPACE
060F 4C0306 0220        JMP  BEGIN         ;JUMP TO BEGIN, REPEAT MAIN LOOP
            0230 ;
            0240 ;  READ AND INTERPRET THE VALUE RETURNED FROM THE JOYSTICK
            0250 ;
0612 AD7802 0260 JOYSTICK LDA STICK     ;LOAD ACC WITH CONTENTS OF $278
0615 C907   0270        CMP  #$7           ;WAS IT PRESSED TO THE RIGHT?
0617 F00D   0280        BEQ  RIGHT         ;IF YES BRANCH TO RIGHT ROUTINE
0619 C90B   0290        CMP  #$B           ;TO THE LEFT?
061B F017   0300        BEQ  LEFT          ;IF SO BRANCH TO LEFT ROUTINE
061D C90E   0310        CMP  #$E           ;14 FOR UP?
061F F021   0320        BEQ  UP
0621 C90D   0330        CMP  #$D           ;13 FOR DOWN?
0623 F02B   0340        BEQ  DOWN
0625 60     0350        RTS
0626 A558   0360 RIGHT  LDA  LOWSCR        ;GET LOW BYTE OF SCREEN RAM
0628 18     0370        CLC                ;CLEAR THE CARRY BIT
0629 6901   0380        ADC  #$1           ;ADD 1 AND CARRY TO ACC.
062B 8558   0390        STA  LOWSCR        ;UPDATE LOWSCR
062D A559   0400        LDA  HISCR         ;GET HIGH BYTE
062F 6900   0410        ADC  #$00          ;ADD CARRY AND ZERO TO HIGH BYTE
0631 8559   0420        STA  HISCR
0633 60     0430        RTS
0634 A558   0440 LEFT   LDA  LOWSCR        ;GET LOW BYTE OF SCREEN RAM
0636 38     0450        SEC                ;SET THE CARRY BIT
0637 E901   0460        SBC  #$1           ;SUBTRACT 1 AND CARRY
0639 8558   0470        STA  LOWSCR
063B A559   0480        LDA  HISCR         ;GET HIGH BYTE SCREEN RAM
063D E900   0490        SBC  #$00          ;ANYTHING IN CARRY TO SUBTRACT?
063F 8559   0500        STA  HISCR         ;UPDATE HIGH BYTE SCREEN RAM
0641 60     0510        RTS
```

```
0642 A558     0520 UP      LDA  LOWSCR        ;LOAD ACC. WITH LOW BYTE
0644 38       0530         SEC                ;SET THE CARRY BIT
0645 E928     0540         SBC  #$28          ;SUBTACT 40 FROM ACCUMULATOR
0647 8558     0550         STA  LOWSCR
0649 A559     0560         LDA  HISCR
064B E900     0570         SBC  #$00          ;SUBTRACT ZERO AND CARRY
064D 8559     0580         STA  HISCR
064F 60       0590         RTS
0650 A558     0600 DOWN    LDA  LOWSCR        ;GET LOW BYTE OF SCREEN RAM
0652 18       0610         CLC                ;CLEAR THE CARRY
0653 6928     0620         ADC  #$28           ;ADD 40 ($28) FOR EACH LINE DOWN
0655 8558     0630         STA  LOWSCR
0657 A559     0640         LDA  HISCR          ;GET HIGH BYTE SCREEN RAM
0659 6900     0650         ADC  #$00           ;ADD ANY CARRY
065B 8559     0660         STA  HISCR          ;UPDATE HIGH BTYE
065D 60       0670         RTS
              0680 ;
              0690 ;  DRAW READS CHAR DATA AND PLACES LINES
              0700 ;  ON SCREEN IN ORDER OF SEQUENCE TO APPEAR LIKE
              0710 ;  A SPINNING PINWHEEL
              0720 ;
065E A200     0730 DRAW    LDX  #$00           ;SET INDEX TO 0
0660 A000     0740         LDY  #$00           ;INDEX
0662 BD7406   0750 NEXTCHR LDA  CHAR,X         ;INDEXED ADDRESSING
0665 9158     0760         STA  (LOWSCR),Y     ;INDEXED INDIRECT ADDRESSING
0667 8A       0770         TXA                 ;TRANSFER X TO ACC.
0668 48       0780         PHA                 ;PUSH ACC. ONTO STACK
0669 207F06   0790         JSR  DELAY          ;JUMP TO DELAY ROUTINE
066C 68       0800         PLA                 ;PULL ACC. OFF STACK
066D AA       0810         TAX                 ;TRANSFER ACC. TO X REG.
066E E8       0820         INX                 ;INCREMENT X
066F E004     0830         CPX  #$4            ;4 LINES IN PINWHEEL
0671 D0EF     0840         BNE  NEXTCHR        ;GET NEXT ONE
0673 60       0850         RTS
0674 7C       0860 CHAR    .BYTE 124,15,13,60  ;VALUES FOR LINES
0675 0F
0676 0D
0677 3C
              0870 ;
              0880 ;  ERASE PUTS A SPACE OVER THE SPINNING
              0890 ;  PINWHEELS LAST POSITION
              0900 ;
0678 A000     0910 ERASE   LDY  #$00           ;INDEX FOR ZERO PAGE ADDRESSING
067A A900     0920         LDA  #$00           ;VALUE FOR SPACE
067C 9158     0930         STA  (LOWSCR),Y     ;STORE IN LAST LOCATION
067E 60       0940         RTS
              0950 ;
              0960 ;  DELAY HOLDS THE IMAGE IN ONE PLACE MOMENTARILY
              0970 ;  BEFORE READING NEXT MOVE
              0980 ;
067F A219     0990 DELAY   LDX  #$19         ;COUNT 0-255 25 TIMES
0681 A000     1000 AGAIN   LDY  #$00
```

```
0683 C8      1010 WAIT   INY                 ;INCREMENT Y REGISTER
0684 D0FD    1020        BNE   WAIT              ;IF NOT ZERO, WAIT
0686 CA      1030        DEX                     ;25 YET?
0687 D0F8    1040        BNE   AGAIN             ;IF NOT ZERO, AGAIN
0689 60      1050        RTS
```

INTERNAL CHARACTER SET

Column 1				Column 2				Column 3				Column 4			
#	CHR	#	CHR	#	CHR	#	CHR	#	CHR	#	CHR	#	CHR	#	CHR
0	Space	16	0	32	@	48	P	64	♥	80	♣	96	♦	112	p
1	!	17	1	33	A	49	Q	65	├	81	┌	97	a	113	q
2	"	18	2	34	B	50	R	66	▕	82	─	98	b	114	r
3	#	19	3	35	C	51	S	67	┘	83	┼	99	c	115	s
4	$	20	4	36	D	52	T	68	┤	84	●	100	d	116	t
5	%	21	5	37	E	53	U	69	┐	85	▄	101	e	117	u
6	&	22	6	38	F	54	V	70	╱	86	▎	102	f	118	v
7	'	23	7	39	G	55	W	71	╲	87	┬	103	g	119	w
8	(	24	8	40	H	56	X	72	◢	88	┴	104	h	120	x
9	)	25	9	41	I	57	Y	73	▗	89	▌	105	i	121	y
10	•	26	:	42	J	58	Z	74	◣	90	└	106	j	122	z
11	+	27	;	43	K	59	[	75	▝	91	[1] ␛	107	k	123	♠
12	,	28	<	44	L	60	\	76	▘	92	↑	108	l	124	|
13	—	29	=	45	M	61	]	77	▔	93	↓	109	m	125	[1] ↰
14	—	30	>	46	N	62	Λ	78	▁	94	←	110	n	126	[1] ◀
15	/	31	?	47	O	63	—	79	▖	95	→	111	o	127	[1] ▶

[1] In mode 0 these characters must be preceded with an escape, CHR$(27), to be printed.